

Martin Glinz

Software Engineering I

Vorlesungsskript, WS 2001/2002

Inhalt

1. Einführung: Software-Entwicklung als Problem
2. Zielsetzung, Messung
3. Der Software-Prozess
4. Software-Projektführung
5. Software-Aufwandschätzung
6. Konzipieren von Lösungen
7. Spezifikation von Anforderungen
8. Realisierung
9. Qualitätsmanagement
10. Dokumentation
11. Konfigurationsverwaltung
12. Produktivitätsfaktoren
- Literatur

Vorbemerkung

Dieses Skript ist als Grundlage für eine zweistündige Einführungsvorlesung in das Gebiet des Software Engineerings an der Universität Zürich konzipiert.

Es wurde aus den Unterlagen zu verschiedenen Vorlesungen und Kursen des Verfassers zusammengestellt. Teile der Unterlagen (insbesondere die Kapitel 1 bis 5) sind neu erstellt oder systematisch überarbeitet worden; andere wurden übernommen und nur in der äußeren Form angepasst. Die Darstellung ist daher nicht ganz homogen. Es ist vorgesehen, mit der Zeit alle Kapitel im gleichen Stil wie die Kapitel 1 bis 5 zu gestalten.

Zürich, im Dezember 1996

Mit wenigen Fehlerkorrekturen und einer Modifikation in Kapitel 5 neu aufgelegt für das WS 1997/98

Zürich, im September 1997

Die 1996 geplante Überarbeitung der Kapitel 6-12 konnte bisher nur teilweise realisiert werden. Das Kapitel 6 wurde vollständig erneuert und erheblich erweitert. In Kapitel 7 wurden die Definitionen überarbeitet und ein neuer Abschnitt über Anwendungsfälle hinzugefügt. In einigen Kapiteln wurden Aufgaben hinzugefügt und die Darstellung redaktionell bearbeitet. In allen Kapiteln wurden verschiedene Details korrigiert und ergänzt. Ferner ist die Rechtschreibung den neuen Regeln angepasst worden.

Zu diesem Skript ist ein Lernzielkatalog erhältlich, aus dem ersichtlich ist, welche Teile des Skripts Bestandteil der Vorprüfung in Informatik für Studierende der Wirtschaftsinformatik und der Ökonomie an der Universität Zürich sind.

Zürich, im September 1998

Durchgesehene Neuauflage für das WS 1999/2000.

Zürich, im August 1999

Durchgesehene, geringfügig ergänzte Neuauflage für das WS 2000/2001. Die Ergänzungen betreffen die Kapitel 2 und 5 sowie das Literaturverzeichnis.

Zürich, im September 2000

Durchgesehene, geringfügig ergänzte Neuauflage für das WS 2001/2002. Die Ergänzungen betreffen die Kapitel 3 (neue Aufgabe 3.1) und 5 (Regel 5.3).

Zürich, im September 2001

Martin Glinz

© 1996 – 2001 by Martin Glinz.

Alle Rechte vorbehalten. Reproduktion zum nicht kommerziellen Gebrauch mit Quellenangabe gestattet. Reproduktion – auch auszugsweise – zum kommerziellen Gebrauch nur mit schriftlicher Bewilligung des Verfassers gestattet.

1. Einführung: Software-Entwicklung als Problem

Software ist in den vergangenen Jahrzehnten zu einem derjenigen Faktoren geworden, ohne die in den hochentwickelten Ländern nichts mehr geht. Die Entwicklung von Software wird jedoch bis heute nur ungenügend beherrscht. Da die Softwarekosten den Hardwarekosten längst den Rang abgelaufen haben (Bild 1.1) und weltweit horrenden Summen für Software ausgegeben werden (Bild 1.2), ist dies ein sehr unbefriedigender Zustand.

In diesem Kapitel wird geklärt, was eigentlich Software ist und warum Software Entwicklung schwierig ist. Auf diesem Hintergrund werden die Idee, die Ziele und die grundlegenden Mittel des Software Engineerings motiviert und eingeführt.

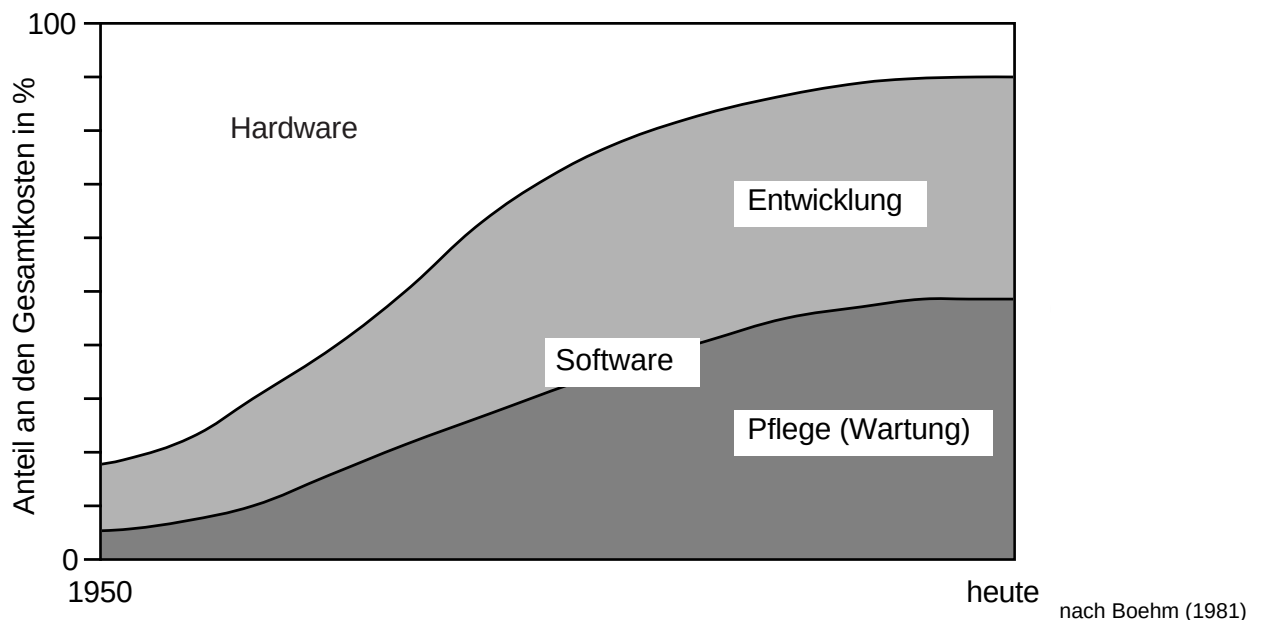


BILD 1.1. Entwicklung des Kostenverhältnisses von Software und Hardware

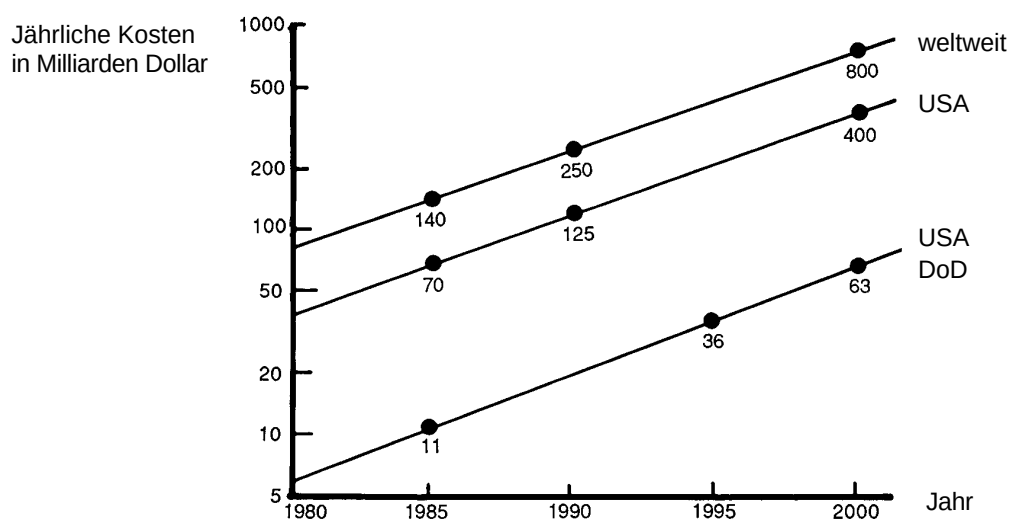


BILD 1.2. Tendenzen im Wachstum der Software-Kosten

1.1 Software

1.1.1 Die Rolle der Software

Rechner durchdringen alle Lebensbereiche – und mit ihnen die Software, welche diese Rechner steuert. Wirtschaft und Gesellschaft sind abhängig geworden von Software. Und die Abhängigkeit nimmt zu. Viele Selbstverständlichkeiten des Alltagslebens sind ohne Rechner und deren Software nicht mehr möglich.

In krassem Gegensatz zu dieser Abhängigkeit steht die Tatsache, dass weltweit die Erstellung von Software nur ungenügend beherrscht wird. Termin- und Kostenüberschreitungen bei Software-Projekten sind die Regel; Software, die Fehler enthält oder sich nicht entsprechend den Vorstellungen und Bedürfnissen der Benutzenden verhält, gehört zum Alltag. Immer wieder kommt es auch vor, dass ganze Projekte scheitern.

Stellvertretend seien zwei spektakuläre Beispiele aus jüngerer Zeit genannt: das Scheitern des CONFIRM-Projekts und der missglückte Erstflug der Ariane 5-Rakete.

Im CONFIRM-Projekt sollte im Auftrag großer amerikanischer Hotel- und Mietwagenunternehmen ein neues, umfassendes Reservierungssystem entwickelt werden. Nach dreieinhalb Jahren Entwicklungszeit und Investitionen von rund 125 Millionen Dollar wurde das Projekt im Juli 1992 eingestellt, als erkannt wurde, dass die gestellten Anforderungen mit dem gewählten Lösungsansatz nicht erreichbar waren (Oz, 1995).

Bei der Ariane 5 führten eine nicht behandelte Ausnahmebedingung in der Software der Trägheitsnavigationssysteme sowie die Fehlinterpretation der Meldung über den Ausfall dieser Systeme als Messdaten(!) durch die Software des Hauptrechners zu unsinnigen Befehlen an die Steuerdüsen und damit zur Zerstörung der Rakete (Lions, 1996).

Dabei erscheint Programmieren auf den ersten Blick gar nicht so schwierig. Was ist ein Programm denn eigentlich mehr als das Zusammensetzen einfacher Befehle zu einer geeigneten Folge von Anweisungen an einen Rechner? Was ist denn Software eigentlich und was hat sie für besondere Eigenschaften, die den Umgang mit ihr so schwierig machen?

1.1.2 Software

DEFINITION 1.1. *Software.* Die Programme, Verfahren, zugehörige Dokumentation und Daten, die mit dem Betrieb eines Computersystems zu tun haben (IEEE 610.12).

Diese Definition erlaubt uns, drei wichtige Feststellungen über Software zu machen, die wesentliche Hinweise darauf geben, warum die Entwicklung von Software schwierig ist.

FESTSTELLUNG¹⁾ 1.1. Software umfasst erheblich mehr als nur Programme (Bild 1.3).

Der Aufwand für das Schreiben der Programme beträgt in der Regel nur 10-20 Prozent des Gesamtaufwands für die Entwicklung von Software. Bezogen auf die gesamten Lebensdauer einer Software sind es unter 10% der Kosten (Bild 1.4). Da die Programme jedoch der entscheidende Bestandteil von Software sind (ohne Programme geht nichts), wird bei der Schätzung des

¹⁾ Wir formulieren in diesem Text alle Gesetzmäßigkeiten über Software und Software Engineering als *Feststellungen* und *Regeln*. Die Wahl dieser Terminologie hat folgenden Hintergrund: In allen Lebensbereichen gilt, dass wir Gesetzmäßigkeiten vermuten, wenn wir gewisse Erfahrungen in gleicher Weise immer wieder machen. In den Wissenschaften wird dann versucht, solche Vermutungen durch Experimente und statistische Verfahren zu untermauern. Auf diese Weise gewinnen wir Gesetze, zum Beispiel das Gravitationsgesetz oder das Hebelgesetz in der Physik. Solche Gesetzmäßigkeiten kennen wir auch für Software. Im Gegensatz zu den Gesetzen der Naturwissenschaften sind die Software-Gesetzmäßigkeiten jedoch nur ansatzweise quantitativ gefasst und statistisch abgesichert. Wir sprechen daher im Folgenden von *Feststellungen*, wenn wir sichere oder empirisch erhärtete *Beobachtungen* beschreiben, und von *Regeln*, wenn wir entsprechende *Zusammenhänge* von Phänomenen beschreiben.

Aufwands für die Entwicklung von Software allzuoft nur auf die Programme abgestellt. Dadurch wird der tatsächliche Aufwand um ein Vielfaches unterschätzt.

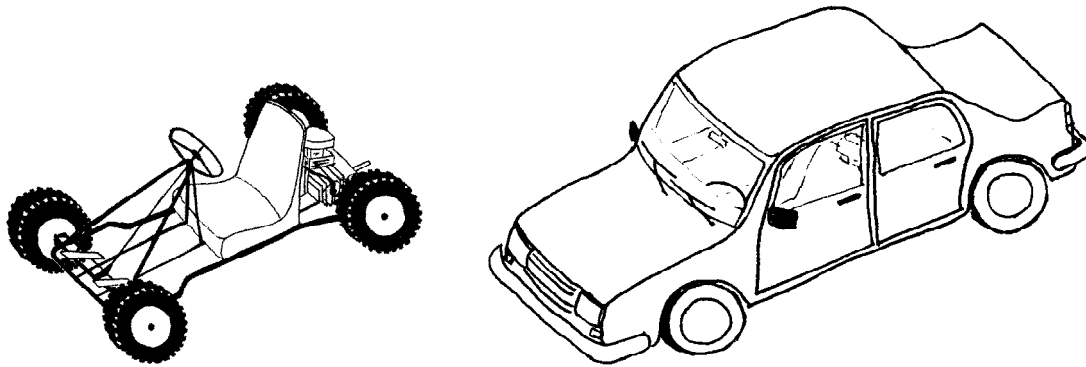
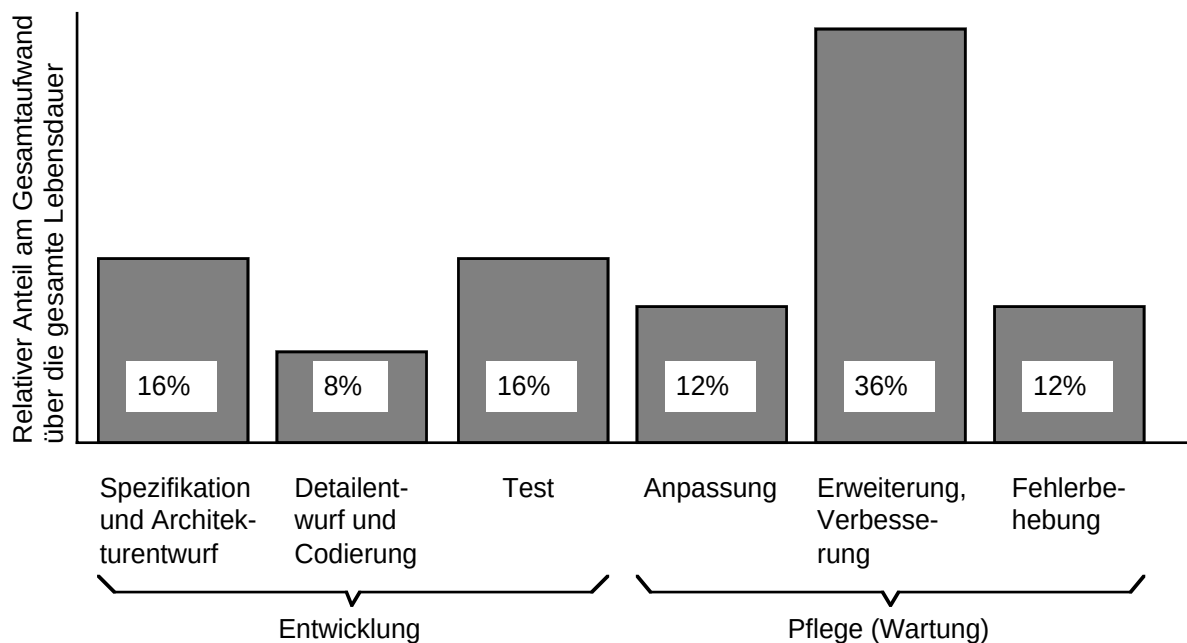


BILD 1.3. So wie ein Auto wesentlich mehr ist als nur ein fahrbarer Untersatz, so umfasst Software wesentlich mehr als nur Programme.



nach Boehm (1981)

BILD 1.4. Verteilung des Aufwands über die Lebensdauer eines Software-Produkts

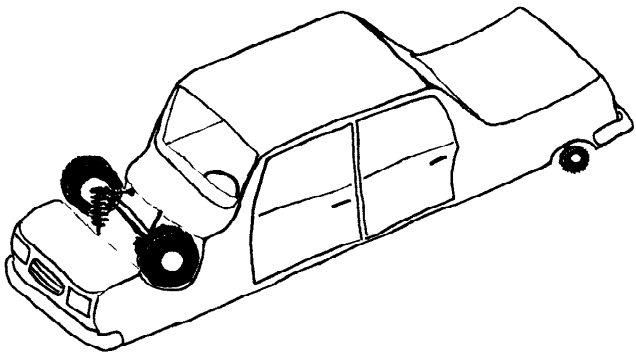
FESTSTELLUNG 1.2. Software ist ein immaterielles technisches Produkt. Man kann Software nicht anfassen.

Die Konsequenzen dieser trivial scheinenden Feststellung sind vielfältig:

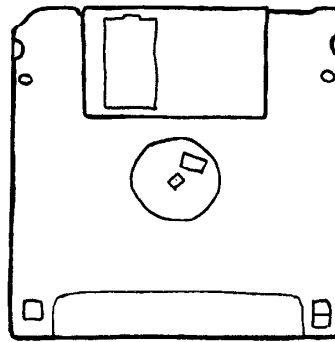
- Geistige Güter werden in unserer Gesellschaft in der Regel als *weniger wertvoll* eingestuft als mit dem gleichen Aufwand erstellte materielle Güter: Der Wert von Software wird *unterschätzt*.
- Im Gegensatz zu materiellen Produkten gibt es für Software keine natürlichen Grenzen in Form von Materialeigenschaften oder Naturgesetzen. Es gibt einzig die theoretische Grenze der Berechenbarkeit, aber diese wirkt sich in der Praxis kaum aus. Während ein Brückenbauingenieur bei seinen Konstruktionen Rücksicht nehmen muss auf die Eigenschaften der verwendeten Materialien und auf die Gesetze der Statik und Schwingungsdynamik, ist ein Software-Entwickler grundsätzlich in der Wahl seiner Konstruktionen frei. Software findet ihre praktischen Grenzen nur in der Begrenztheit des Könnens ihrer Entwickler. Da diese Grenzen sehr

weit gesteckt sind und Menschen außerdem dazu neigen, die Begrenztheit ihres Könnens zu verdrängen, werden bei Software häufiger und leichtsinniger (zu) schwierige Vorhaben angegangen als in anderen technischen Disziplinen.

- Bei der Entwicklung materieller Produkte sind viele *Fehler* leicht als solche erkennbar. Wenn beim Bauen das Dach statt des Kellers in die Baugrube gesetzt wird, so ist dies unschwer als Fehler zu erkennen. Ein Fehler ähnlicher Schwere in Software ist nur durch sorgfältige und aufwendige Prüfmaßnahmen zuverlässig erkennbar oder verhütbar (Bild 1.5).
- Gleiches gilt für die Beurteilung des *Entwicklungsstands*: Behauptet ein Bauunternehmen, der Bau sei praktisch fertig, obwohl erst der Rohbau steht, so ist leicht feststellbar, dass diese Behauptung nicht stimmt. Die Beurteilung des Fertigstellungsgrads von Software ist ungleich schwieriger und aufwendiger.
- Software ist scheinbar sehr flexibel und leicht zu ändern. Es gibt ja kein Material, das dabei umgeformt oder gar neu produziert werden muss. Kleinst-Software ist tatsächlich leicht änderbar. Mit zunehmender Größe der Software hat jede Änderung jedoch so viele Effekte und Konsequenzen, die alle bedacht sein müssen, dass die tatsächlichen Aufwendungen für Software-Änderungen oft höher sind als diejenigen für vergleichbare materielle Produkte.



Diese mechanische Konstruktion ist offensichtlich falsch.



Wie erkennen wir aber Fehler in der hier gespeicherten Software-Konstruktion?

BILD 1.5. Software kann man nicht anfassen.

FESTSTELLUNG 1.3. Software verhält sich (im mathematischen Sinn) unstetig.

Software ist Bestandteil eines digital arbeitenden Rechnersystems. Solche Systeme haben die unangenehme Eigenschaft, dass kleinste Veränderungen in Programmen oder Daten (Im Extremfall genügt schon die Setzung eines Punkts anstelle eines Kommas) massive Veränderungen im Verhalten des Systems bewirken können. Es ist daher ungleich schwieriger und aufwendiger, das wunschgemäße Funktionieren von Software mit einer gegebenen Wahrscheinlichkeit zu gewährleisten, als dies bei Produkten der klassischen technischen Disziplinen der Fall ist.

1.1.3 Wozu dient Software?

Software dient dazu, ein Problem zu lösen oder zu dessen Lösung beizutragen, indem menschliche oder technische Arbeitsvorgänge automatisiert oder unterstützt werden. Software steht daher in einer ständigen Wechselwirkung mit Arbeits- und Produktionsprozessen und mit den daran beteiligten Menschen. Daraus folgen drei weitere wesentliche Eigenschaften von Software.

FESTSTELLUNG 1.4. Wenn ein Problem von seiner Natur her komplex und schwierig zu lösen ist, so ist die Software zur Lösung dieses Problems in der Regel nicht weniger komplex und schwierig.

FESTSTELLUNG 1.5. Bei der Lösung von Problemen mit Software sind immer zwei Schwierigkeiten gleichzeitig zu bewältigen: (1) Das Problem ist im Kontext seines Sachgebiets zu verstehen und befriedigend zu lösen. (2) Die Problemlösung muss auf adäquate Software-Strukturen abgebildet werden.

FESTSTELLUNG 1.6. Problemlösungen schaffen neue Realitäten und wecken neue Bedürfnisse.

Software ist daher nicht einfach ein Abbild der Realität und bisheriger manueller Problemlösungen. Sie *konstruiert und verändert* die Realität. Beispielsweise ermöglicht Logistik-Software die Fertigung von Gütern nach dem "Just in time"-Prinzip (Zulieferteile werden genau dann angeliefert, wenn sie in der Produktion benötigt werden). Dies schafft streng termingebundenen Lastwagenverkehr als neue Realität und softwaregestützte Optimierung solchen Verkehrs als neues Bedürfnis.

1.2 Software-Entwicklung

1.2.1 Definition

Software-Entwicklung umfasst alle Tätigkeiten und Ressourcen, die zur Herstellung von Software notwendig sind.

DEFINITION 1.2. *Software-Entwicklung.* Die Umsetzung der Bedürfnisse von Benutzern in Software. Umfasst Spezifikation der Anforderungen, Konzept der Lösung, Entwurf und Programmierung der Komponenten¹⁾, Zusammensetzung der Komponenten und ihre Einbindung in vorhandene Software, Inbetriebnahme der Software sowie Überprüfung des Entwickelten nach jedem Schritt.

1.2.2 Einige grundlegende Gesetzmäßigkeiten

In diesem Abschnitt zeigen wir einige grundlegende Gesetzmäßigkeiten für die Entwicklung von Software, die wesentlich dazu beitragen, dass Software-Entwicklung etwas Schwieriges ist.

FESTSTELLUNG 1.7. Die Entwicklung von Klein-Software unterscheidet sich fundamental von der Entwicklung größerer Software.

Letzteres stellt dabei den Normalfall dar. Tabelle 1.1 charakterisiert die Unterschiede. Viele Probleme existieren für Klein-Software gar nicht. Die Erfahrung, dass die Entwicklung kleiner Programme recht einfach ist, ist der Hauptgrund für den Trugschluss vieler Leute, dass Software-Entwicklung generell etwas Einfaches sein müsse.

REGEL 1.1. Der Aufwand für die Erstellung von Software steigt mit wachsender Produktgröße überproportional an. Nach Boehm (1981) wird der Zusammenhang beschrieben durch eine Gleichung der Art $A = bP^m$

Dabei ist A der Aufwand, P die Größe der Software, b ein konstanter Faktor und m ein Exponent im Bereich zwischen 1,05 und 1,2.

Die Regel 1.1 ist eine der wenigen, welche empirisch erhärtet sind (vgl. Boehm (1981) und Kapitel 5).

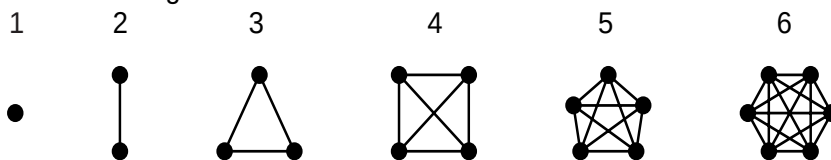
¹⁾ Dabei wird unter «Programmieren» sowohl das Schreiben neuer Programme wie auch das Erweitern oder Modifizieren vorhandener Programme verstanden.

Ursachen für dieses überproportionale Wachstum sind unter anderem, dass der Aufwand zur Beherrschung der wachsenden Komplexität und der Kommunikationsaufwand überproportional wachsen. Das Wachstum einzelner Faktoren ist dabei nicht kontinuierlich, sondern weist Quantensprünge auf (Bild 1.6).

TABELLE 1.1. Klein-Groß-Gegensätze in der Software-Entwicklung

klein	groß
Programme von 1 bis ca. 300 Zeilen Länge	Längere Programme
Für den <i>Eigengebrauch</i>	Für den Gebrauch durch <i>Dritte</i>
<i>Vage Zielsetzung</i> genügt, das Produkt ist seine eigene Spezifikation	<i>Genaue Zielbestimmung</i> , d.h. die Spezifikation von Anforderungen, erforderlich
<i>Ein Schritt</i> vom Problem zur Lösung genügt: die Lösung wird direkt programmiert	<i>Mehrere Schritte</i> vom Problem zur Lösung erforderlich: Spezifikation der Anforderungen, Konzept der Lösung, Entwurf der Teile, Programmieren der Teile, Zusammensetzen der Teile, Inbetriebnahme
<i>Validierung</i> (Feststellen, ob die Software sich den Erwartungen entsprechend verhält) und nötige Korrekturen finden am <i>Endprodukt</i> statt	<i>Auf jeden Entwicklungsschritt muss ein Prüfschritt</i> folgen, sonst wird das Risiko, dass das Endergebnis unbrauchbar ist, viel zu groß
<i>Eine Person</i> entwickelt: Keine Koordination mehrerer beteiligter Personen erforderlich, keine Kommunikationsbedürfnisse	<i>Mehrere Personen</i> entwickeln gemeinsam: Koordination und Kommunikation notwendig
<i>Komplexität</i> des Problems in der Regel <i>klein</i> , Strukturieren der Software und Behalten der Übersicht nicht schwierig	<i>Komplexität</i> des Problems <i>größer bis sehr groß</i> , explizite Maßnahmen zur Strukturierung und Modularisierung erforderlich
Software besteht aus <i>wenigen Komponenten</i>	Software besteht aus <i>vielen Komponenten</i> , die spezielle Maßnahmen zur Komponentenverwaltung erfordern
In der Regel wird <i>keine Dokumentation</i> erstellt	<i>Dokumentation</i> dringend <i>erforderlich</i> , damit Software wirtschaftlichen betrieben und gepflegt werden kann
<i>Keine Planung und Projektorganisation</i> erforderlich	<i>Planung und Projektorganisation</i> <i>zwingend</i> erforderlich für eine zielgerichtete, wirtschaftliche Entwicklung

Anzahl beteiligte Personen



Anzahl Kommunikationspfade

0 1 3 6 10 15



Quantensprung:
Kommunikation
wird erforderlich



Quantensprung: Zahl der
Kommunikationspfade über-
steigt Zahl der Personen

BILD 1.6. Wachstum und Quantensprünge beim Kommunikationsbedarf

FESTSTELLUNG 1.8. Software ist einer Evolution unterworfen (vgl. Kapitel 3.1).

Die Umwelt und die Bedürfnisse der Benutzenden verändern sich ständig. Software, die sich in Gebrauch befindet, bleibt daher ohne ständige Anpassungen und Erweiterungen nicht gebrauchstauglich. Ferner werden im Betrieb auch immer wieder Fehler entdeckt, die behoben werden müssen. Solche Entwicklungsarbeiten an in Betrieb befindlicher Software werden *Pflege* oder *Wartung* genannt.

Das Tempo der Software-Evolution ist so hoch, dass häufig schon während der Entwicklung die Entwicklungsziele an veränderte Bedürfnisse angepasst werden müssen.

Die erforderlichen Anpassungen und Erweiterungen bei der Pflege bringen es mit sich, dass Umfang und innere Unordnung von in Gebrauch befindlicher Software ständig zunehmen. Die Pflege einer Software wird daher mit zunehmendem Alter immer schwieriger und teurer. Dies ist ein Effekt, den man auch in anderen Disziplinen beobachten kann. Wird zum Beispiel ein Haus über Jahrzehnte hinweg ständig erweitert und umgebaut, so wird es immer größer und unübersichtlicher. Je planloser und je häufiger an- oder umgebaut wird, desto schwieriger ist es, den Bau so auszuführen, dass nicht Teile des Gebäudes dabei einstürzen oder danach ihren Verwendungszweck nicht mehr erfüllen.

FESTSTELLUNG 1.9. Software wird von Menschen gemacht.

Die Fähigkeiten dieser Menschen ebenso wie ihre emotionalen Einstellungen und alle ihre Unzulänglichkeiten haben einen direkten und erheblichen Einfluss auf die von ihnen entwickelte Software (vgl. DeMarco und Lister 1991, Glinz 1988 und Weinberg 1971).

1.2.3 Warum ist Software-Entwicklung schwierig?

Wenn wir obenstehende Feststellungen und Regeln und die ihnen zugrundeliegenden Phänomene analysieren, so stellen wir fest, dass es Wesentlichen vier Faktoren sind, welche die Entwicklung von Software schwierig machen. Es sind dies

- Die Größe der zu lösenden Probleme. Software ist nicht einfacher, als die Probleme, die sie löst. Je größer und schwieriger die Software, desto aufwendiger und schwieriger ist ihre Entwicklung. Fortschritte bei der Beherrschung des Software-Entwicklungsprozesses werden durch das Angehen größerer und schwierigerer Probleme immer wieder kompensiert.
- Die Tatsache, dass Software ein immaterielles Produkt ist. Die Immaterialität macht das Arbeiten mit Software schwieriger als dasjenige mit materiellen technischen Produkten vergleichbarer Komplexität. Die Risiken sind schwieriger zu erkennen; ad-hoc-Vorgehensweisen führen schneller ins Desaster.
- Sich permanent verändernde Ziele aufgrund der Evolution. Schon das Bestimmen und Erreichen fixierter Ziele bei der Entwicklung eines Produkts ist keine leichte Aufgabe. Sich verändernde Ziele machen das Ganze noch um eine Größenordnung schwieriger.
- Fehler infolge von emotionalen Fehleinschätzungen (Immaterielles hat keinen Wert und ist total flexibel, was im Kleinen geht, geht genauso im Großen, etc.). Software-Entwicklung wird daher unbewusst-emotional meist als viel einfacher eingeschätzt, als sie tatsächlich ist. Dies führt zu unrealistischen Erwartungen und zu von Beginn weg zu tiefen Kosten- und Terminschätzungen. Eine besonders häufige Fehlerursache ist das Denken und Handeln in der Welt von Klein-Software, während tatsächlich große Software zu entwickeln ist (vgl. Tabelle 1.1). Besonders verheerend wirkt sich die lineare Extrapolation von Erfahrungen mit Klein-Software auf große Software aus (Bild 1.7).

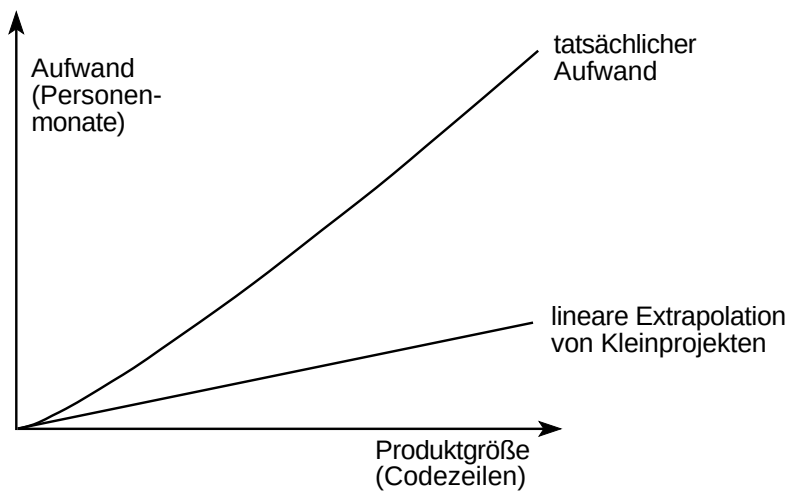


BILD 1.7. Fehleinschätzungen bei linearer Extrapolation

1.3 Software Engineering

In der Anfangszeit der Informatik gab es mit der Entwicklung von Software kaum Probleme. Dies ist nicht weiter verwunderlich, denn die Software bestand aus einzelnen, weitgehend voneinander unabhängigen Programmen, die jedes für sich eine beherrschbare Größe hatten. Die Schwierigkeiten der Software-Entwicklung manifestierten sich erst in den 60er Jahren, als immer umfangreichere Software entwickelt wurde und die bis anhin verwendeten ad-hoc-Vorgehensweisen zunehmend versagten. Zur Überwindung dieser *Software-Krise* erhob Friedrich Ludwig Bauer 1968 die Forderung nach *Software Engineering*, d.h. einem systematischen Vorgehen bei der Entwicklung, wie es in klassischen Ingenieurdisziplinen seit langem üblich ist.

Dabei muss man wissen, dass zur damaligen Zeit allgemein die Auffassung herrschte, das Schreiben von Software sei eine Kunst und erfordere daher individuelle künstlerische Freiheit für die Programmierer. Auf diesem Hintergrund gesehen war Bauers Forderung revolutionär.

In der Zwischenzeit hat sich die Erkenntnis weitestgehend durchgesetzt, dass bei nichttrivialen Aufgaben eine wirtschaftliche und termintreue Entwicklung qualitativ guter Software ohne Software Engineering nicht möglich ist.

Trotz aller seither erzielter Fortschritte ist uns die Software-«Krise» treu geblieben. Das hat verschiedene Gründe. Einerseits zeigt sich, dass gegen gesicherte Erkenntnisse des Software Engineerings immer wieder verstoßen wird. Die Ursachen dafür liegen (neben Unkenntnis) meistens in emotional bedingten Fehleinschätzungen. Andererseits werden Fortschritte dazu benutzt, immer größere (und schwierigere) Software-Systeme zu entwickeln (vgl. Abschnitt 1.2.3).

Rückblickend wird klar, dass es sich um keine Krise gehandelt hat, sondern um die erste Manifestation der grundsätzlichen Schwierigkeit der Entwicklung großer Software.

Die von Fairley (1985) gegebene Definition gibt das heutige Verständnis von Software Engineering sehr gut wieder:

DEFINITION 1.3. *Software Engineering* ist das technische und planerische Vorgehen zur systematischen Herstellung und Pflege von Software, die zeitgerecht und unter Einhaltung der geschätzten Kosten entwickelt bzw. modifiziert wird.

Das IEEE Standard Glossary of Software Engineering Terminology (IEEE 610.12, 1990) nimmt das quantitative, auf gemessenen Größen basierende Arbeiten als weiteres Kriterium hinzu.

DEFINITION 1.4. Software Engineering. Die Anwendung eines systematischen, disziplinierten und quantifizierbaren Ansatzes auf die Entwicklung, den Betrieb und die Wartung von Software, das heißt, die Anwendung der Prinzipien des Ingenieurwesens auf Software (IEEE 610.12).

Im deutschen Sprachraum ist anstelle von Software Engineering auch der Terminus *Softwaretechnik* gebräuchlich.

1.4 Ziele und Mittel des Software Engineerings

Mit Software Engineering werden drei grundsätzliche Ziele verfolgt.

Die *Produktivität* bei der Herstellung von Software soll gesteigert werden. Angesichts der horrenden Summen, die jährlich weltweit für Software ausgegeben werden (Bild 1.2), sind auch kleine Produktivitätssteigerungen von großem wirtschaftlichem Interesse.

Die *Qualität* der erstellten Software soll verbessert werden. Dies bringt neben Wettbewerbsvorteilen durch zufriedene Kunden auch Kostensenkungen bei der Pflege und Weiterentwicklung von in Betrieb befindlicher Software.

Die *Führbarkeit* von Software-Entwicklungsprojekten soll erleichtert werden. Dies bringt Erleichterungen durch bessere Termin- und Kostentreue und erleichtert die Kontrolle der Projektrisiken. Letztlich bedeutet dies Senkung der Kosten bei verbesserter Qualität.

Die Mittel, die zur Erreichung dieser Ziele zur Verfügung stehen, sind in Bild 1.8 skizziert. Alle folgenden Kapitel sind der näheren Beschreibung dieser Mittel gewidmet.

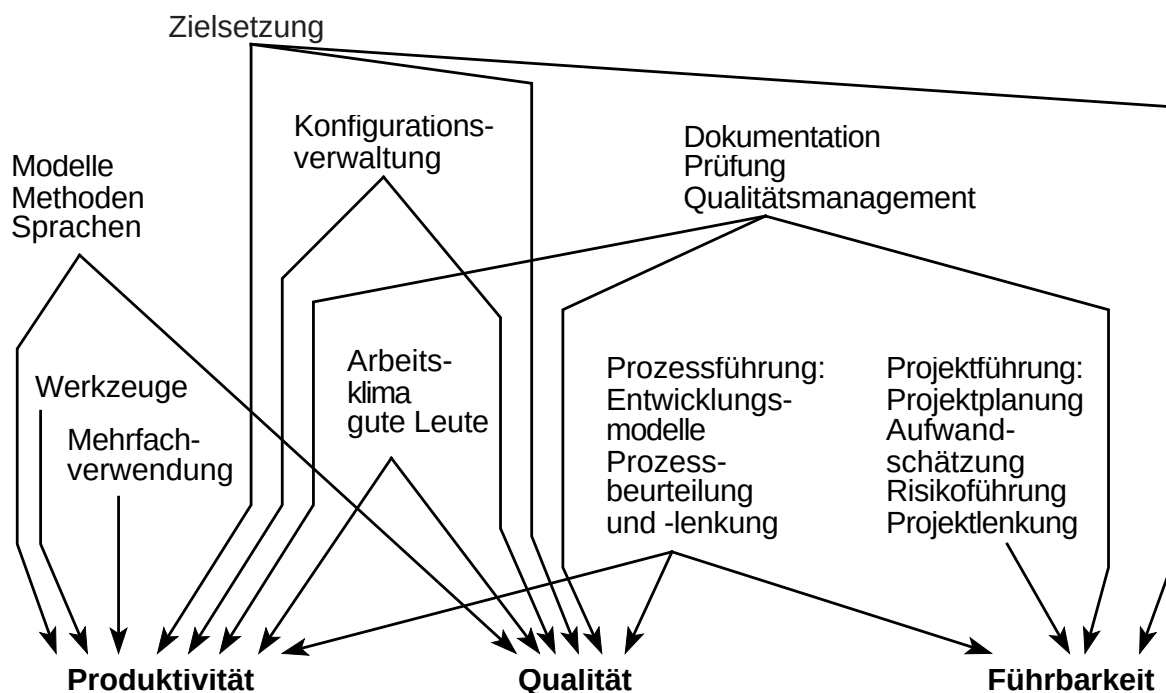


BILD 1.8. Mittel des Software Engineerings und ihre Wirkung

Aufgaben

- 1.1** Begründen Sie die vier Hauptschwierigkeiten bei der Entwicklung von Software aus den Feststellungen 1.1 bis 1.9 und der Regel 1.1.

- 1.2** «Ein Mann braucht zum Bau einer 2 m langen Brücke 0,5 Tage. Wie lange brauchen 100 Leute für den Bau einer 2 km langen Brücke? Rechne.»
Begründen Sie, warum das eine Milchmädchenrechnung ist. Ziehen Sie Parallelen zur Entwicklung von Software.
- 1.3** Was sind die drei grundlegenden Ziele des Software Engineerings?
- 1.4** Eine Kundenbetreuerin im Firmenkundengeschäft einer Bank hat auf der Grundlage eines Tabellenkalkulationsprogramms eine kleine persönliche Anwendung geschrieben, die sie bei der Überprüfung der Kredite der von ihr betreuten Firmen unterstützt. Die notwendigen Daten gibt sie jeweils von Hand ein. Der Abteilungsleiter sieht diese Anwendung zufällig, ist davon angetan und beschließt, sie allen Kundenbetreuerinnen und -betreuer zur Verfügung zu stellen. Die notwendigen Daten sollen neu automatisch aus den Datenbanken der Bank übernommen werden.
Die Kundenbetreuerin gibt an, für die Entwicklung ihrer Anwendung insgesamt etwa vier Arbeitstage aufgewendet zu haben. Der Abteilungsleiter veranschlagt daher für die Übernahme und die gewünschten Änderungen einen Aufwand von einer Arbeitswoche. Als die geänderte Anwendung endlich zur Zufriedenheit aller Beteiligten läuft, sind jedoch rund acht Arbeitswochen Aufwand investiert.
Der Abteilungsleiter erzählt die Geschichte einem befreundeten Berater als Beispiel, dass Informatik-Projekte nie ihre Termine einhalten. Darauf meint der Berater trocken, der investierte Aufwand sei völlig realistisch und normal. Begründen Sie, warum.
- 1.5** Ein Unternehmen stellt Messgeräte her, in denen Messungen und Gerätebedienung mit Hilfe von Software erfolgen. Bei einem in Entwicklung befindlichen neuen Gerät findet die für die Hardware verantwortliche Abteilung heraus, dass pro Gerät 20 Franken eingespart werden können, wenn der geplante Prozessor durch einen primitiveren ersetzt wird. Der Verkauf rechnet damit, während 5 Jahren je etwa 400 Geräte abzusetzen; es kann also eine Einsparung von ca. 40'000 Franken erwartet werden. Die Änderung wird daher beschlossen, kleine Anpassungen in der Software sind ja kein Problem. Als die Softwareentwickler zwei Monate später mit ersten Tests auf dem Zielsystem beginnen wollen, stellt sich heraus, dass ihr bisher immer verwendetes Echtzeit-Betriebssystem auf dem neuen Prozessor nicht läuft und dass der stattdessen angebotene Echtzeit-Kern für ihre Bedürfnisse nicht ausreicht. Durch die Verzögerung und die notwendigen Software-Erweiterungen des Echtzeit-Kerns entstehen Mehrkosten von ca. 100'000 Franken.
Geben Sie Gründe an, wie es zu einer solchen Fehlentscheidung kommt. Wie hätte der Fehler vermieden werden können?

Ergänzende und vertiefende Literatur

Brooks (1995) liefert in einer Sammlung von Essays eine unterhaltsame und lesenswerte Beschreibung von Fehlern im Software Engineering und ihren Ursachen. Das Buch von 1995 ist eine Neuauflage des Originals von 1975 mit zwei neuen Kapiteln.

IEEE 610.12 (1990) ist die Standardreferenz für die englischsprachige Terminologie im Software Engineering. Viele Definitionen in diesem Text sind Übersetzungen aus IEEE 610.12 oder lehnen sich an diese an.

Lehman und Belady (1985) verdanken wir die Erkenntnisse über die Software-Evolution.

McDermid (1991) behandelt alle Gebiete der Informatik mit Bezug zum Software Engineering einschließlich der Grundlagen- und Randgebiete in knappen Übersichtsartikeln in der Art einer Enzyklopädie, aber nach Sachgebieten, nicht nach Stichworten geordnet.

Der Begriff «Software Engineering» wurde von Bauer auf einer NATO-Konferenz in Garmisch-Partenkirchen geprägt. Der Tagungsband dieser Konferenz (Naur, Randell und Buxton 1976) gibt einen Eindruck der damaligen Situation.

Weinberg (1971), DeMarco und Lister (1991) und Glinz (1988) behandeln Probleme der Software-Psychologie, der Menschenführung in der Software-Entwicklung und den Gegensätzen zwischen rationalen und emotionalen Einstellungen im Software Engineering.

Zitierte Literatur

Boehm, B. (1981). *Software Engineering Economics*. , Englewood Cliffs, N.J.: Prentice-Hall.

Boehm, B. (1987). Improving Software Productivity. *IEEE Computer* **20**, 9 (Sept. 1987). 43-57.

Brooks, F.P. (1995). *The Mythical Man Month. Essays on Software Engineering*. Anniversary Edition Reading, Mass., etc.: Addison-Wesley.

DeMarco, T., T. Lister (1991). *Wien wartet auf Dich! Der Faktor Mensch im DV-Management*. München-Wien: Hanser.

Fairley, R. (1985). *Software Engineering Concepts*. New York, etc.: McGrawHill.

Glinz, M. (1988). Emotionales und Rationales im industriellen Software Engineering. *Technische Rundschau* 20/88, 78-81.

IEEE (1990). *Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990. IEEE Computer Society Press.

Lehman, M.M., L.A. Belady (Hrsg.) (1985). *Program Evolution: Processes of Software Change*. London, etc.: Academic Press.

Naur, P., B. Randell, J.N. Buxton (Hrsg.) (1976). *Software engineering: concepts and techniques: proceedings of the NATO conferences*. (Neuausgabe der Proceedings der NATO-Konferenz von 1968 und der Nachfolgekonzferenz von 1969). New York: Petrocelli.

McDermid, J. (1991). *Software Engineer's Reference Book*. Oxford: Butterworth-Heinemann. ca. 1200 p.

Oz, E. (1994). When Professional Standards are Lax. The CONFIRM Failure and its Lessons. *Communications of the ACM* **37**, 10 (Oct 1994). 29-36.

Weinberg, G.M. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.

Lions, J.L. (1996). *ARIANE 5 Flight 501 Failure*. Report by the Inquiry Board. Paris: ESA. <http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html>

2. Zielsetzung, Messung

Ohne definierte *Ziele* ist keine systematische Software-Entwicklung möglich. Eine systematische Zielerreichung ist nur dann möglich, wenn die Ziele kontinuierlich verfolgt und Abweichungen festgestellt werden. Hierzu sind *Messungen* erforderlich.

In diesem Kapitel werden die Problematik der Zielsetzung und Zielverfolgung sowie die Grundlagen der dafür erforderlichen Messtechnik behandelt.

2.1 Das Zielsetzungsproblem

Zielgerichtetes Arbeiten ist eine notwendige Voraussetzung für jegliche Art von systematischer Entwicklung.

In einer Entwicklung ohne formulierte Ziele sind die Eigenschaften und Qualitäten der resultierenden Software zufällig. Sie ergeben sich bestenfalls noch teilweise aus offensichtlichen Bedürfnissen der Problemstellung. Zur Hauptsache hängen sie jedoch dann ab vom Charakter und den Vorlieben der Entwicklerinnen und Entwickler, von Gruppenstrukturen und von den Beziehungen zwischen Entwicklern und Benutzern.

Software Engineering betreiben bedeutet daher zwingend, dass für jede Entwicklung von Software Ziele gesetzt werden und anschließend systematisch auf diese Ziele hin gearbeitet wird.

Es gibt jedoch für die Herstellung von Software kein „natürliches“ Ziel. Viele mögliche Teilziele stehen in Konkurrenz zueinander. Beispielsweise kann eine Software nicht gleichzeitig extrem zuverlässig und besonders kostengünstig in der Entwicklung sein.

Die Wahl der Ziele hat einen erheblichen Einfluss sowohl auf das entstehende Produkt als auch auf den Entwicklungsprozess. Ein von Weinberg und Schulman (1974) durchgeführtes Experiment zeigt dies in eindrucksvoller Weise (Bild 2.1). Weinberg und Schulman hatten fünf Entwicklungsgruppen die gleiche Software entwickeln lassen, den fünf Gruppen aber unterschiedliche Projektziele gesetzt. Das Ergebnis zeigt zweierlei:

- Die Qualitäten der erzeugten Programme sind stark korreliert sind mit den Zielen, welche den Entwicklungsgruppen vorgegeben wurden.
- Die optimale Erreichung des gesetzten Ziels ging zum Teil erheblich auf Kosten anderer Qualitäten. Das heißt, dass manche Ziele sich gegenseitig konkurrenzieren.

Es kann daher keinen für alle Software-Vorhaben tauglichen Satz optimaler Ziele geben, sondern die Ziele müssen für jedes Vorhaben entsprechend den jeweiligen konkreten Bedürfnissen und Randbedingungen neu festgesetzt werden.

Bei der Planung muss berücksichtigt werden, dass manche Ziele abhängig voneinander sind (z.B. Termine und Sachziele) und daher keine beliebigen Freiheitsgrade bei der Zielfestlegung bestehen. Dort, wo wesentliche Ziele sich konkurrenzieren, ist es sinnvoll, die Ziele mit Prioritäten zu versehen.

FESTSTELLUNG 2.1. Es gibt keine natürlichen Ziele für Software. Die Ziele müssen für jedes Entwicklungsvorhaben neu festgelegt werden. Dabei ist zu beachten, dass sich Ziele konkurrenzieren können und dass Ziele voneinander abhängig sein können.

FESTSTELLUNG 2.2. Die Wahl der Ziele hat einen erheblichen Einfluss auf das Produkt und den Entwicklungsprozess.

Ziel: Optimiere...	Qualität der Ergebnisse				
	Erstellungs- aufwand	Anzahl Anweisungen	Speicherbedarf	Klarheit des Programms	Klarheit der Ausgaben
Erstellungs- aufwand	1	4	4	5	3
Anzahl Anweisungen	2 - 3	1	2	3	5
Speicherbedarf	5	2	1	4	4
Klarheit des Programms	4	3	3	2	2
Klarheit der Ausgaben	2 - 3	5	5	1	1

BILD 2.1. Leistungen von Programmierern in Abhängigkeit von den vorgegebenen Zielen.
1 = beste Leistung, 5 = schlechteste Leistung

2.2 Klassifikation von Zielen

In den meisten Software-Projekten lassen sich die Ziele nach dem in Bild 2.2 gezeigten Schema ordnen. Wesentlich ist die Untergliederung der Ziele in Projekt- und Produktziele, wobei die Erreichung der Produktziele mit ein zentrales Projektziel ist. Die Anforderungen an die bereitzustellende Software sind ein Teil der Zielsetzung.

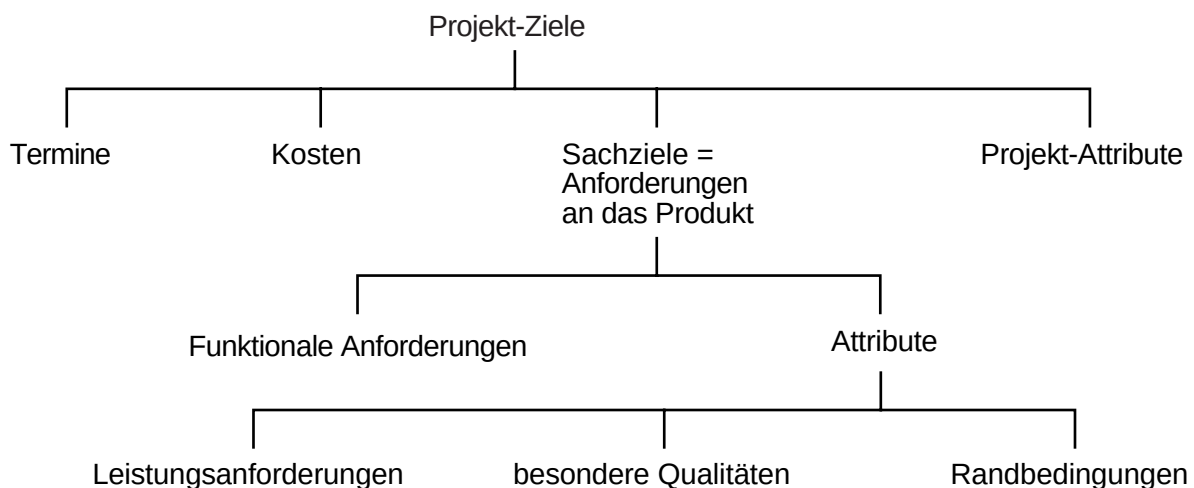


BILD 2.2. Klassifikationsschema für Ziele

2.3 Zielverfolgung

Zielsetzung ist notwendig für ein systematisches Vorgehen, aber nicht hinreichend. Nach der Festlegung der Ziele muss systematisch auf das Erreichen der Ziele hingearbeitet werden. Dabei genügt ein einmaliges Fokussieren auf die Ziele zu Beginn nicht. Langsame, aber stetige Abweichungen von den Zielen während der Entwicklung können dazu führen, dass Ziele massiv verfehlt werden, ohne dass dies rechtzeitig bemerkt wird. Es braucht daher eine regelmäßige Ziel-

kontrolle, so dass bei Abweichungen geeignete Gegenmaßnahmen getroffen werden können. Zielkontrolle ist jedoch nur möglich, wenn das Erreichte mit den Zielen verglichen werden kann.

Es muss daher möglich sein, die Ziele zu *messen*. Hierzu müssen zunächst einmal geeignete Maße gefunden oder definiert werden. Dann müssen für jedes Ziel *Referenzwerte* festgelegt werden. Im Minimum sind dies der Zielwert (wenn dieser Wert gemessen wird, ist das Ziel zu 100% erreicht) und ein Schwellwert (wenn dieser Wert mindestens erreicht ist, wird das Ergebnis als nahe genug beim Ziel akzeptiert).

2.4 Messung

Durch Messungen werden interessierende Merkmale eines Gegenstands quantitativ erfassbar gemacht.

2.4.1 Maße

Ein Maß ordnet mit Hilfe eines *Messverfahrens* einer Menge von Gegenständen *Messwerte* auf einer *Skala* zu. Die Messwerte machen eine Aussage über das zu messende *Merkmal* der gemessenen Gegenstände. Dabei müssen die Menge der Merkmalswerte und die Skala *strukturähnlich* sein, das heißt, die Eigenschaften der Skala müssen mit denen des Merkmals übereinstimmen.

DEFINITION 2.1. *Maß.* Sei D eine Menge gleichartiger Gegenstände mit einem zu messenden Merkmal M , sei $M(d)$ die Ausprägung des Merkmals M für den Gegenstand $d \in D$ und sei M die Menge aller dieser Merkmalsausprägungen. Ein *Maß* für das Merkmal M ist eine Abbildung $\mu: D \rightarrow S$, welche jedem $d \in D$ einen Messwert $\mu(d)$ auf einer *Skala* S so zuordnet, dass M und S strukturähnlich sind. Dies ist genau dann der Fall, wenn für beliebige d_1, d_2 aus D gilt:
Zu jeder Relation R auf der Menge der Merkmalsausprägungen M gibt es eine analoge Relation R^* auf der Skala S mit folgenden Eigenschaften

- (1) $R(M(d_1), M(d_2)) \Rightarrow R^*(\mu(d_1), \mu(d_2))$
- (2) $R^*(\mu(d_1), \mu(d_2)) \Rightarrow R(M(d_1), M(d_2))$ und die Aussage $R(M(d_1), M(d_2))$ ist sinnvoll interpretierbar.

Maße für Software werden in der Literatur oft auch als *Metriken* bezeichnet. Die Bedingungen (1) und (2) werden auch als *Repräsentations-* oder als *Homomorphiebedingung* bezeichnet.

Häufig werden Maße definitorisch eingesetzt, d.h. sie *definieren* ein intuitives Merkmal einer Menge von Gegenständen.

BEISPIEL 2.1. Messung der *Größe* eines Programms. Ein mögliches Maß γ für das Merkmal «Größe» eines Programms besteht aus der Abbildungsvorschrift «Zähle die Programmzeilen, die nicht leer sind oder ausschließlich Kommentar enthalten» und den nicht negativen ganzen Zahlen als Skala.

Auf dem Merkmal «Größe» eines Programms gibt es zwei Relationen: die Merkmalsausprägungen sind *vergleichbar* und sie sind *additiv*. Beide Relationen müssen durch die Abbildungsvorschrift so auf entsprechende Relationen der Skala abgebildet werden, dass die Homomorphiebedingung erfüllt ist; andernfalls wäre γ kein Maß. Es muss also für beliebige Programme P_1, P_2 und P_3 gelten und sinnvoll interpretierbar sein

- (1) $\text{Größe}(P_1) \leq \text{Größe}(P_2) \Rightarrow \gamma(P_1) \leq \gamma(P_2)$
 $\text{Größe}(P_1) + \text{Größe}(P_2) = \text{Größe}(P_3) \Rightarrow \gamma(P_1) + \gamma(P_2) = \gamma(P_3)$
- (2) $\gamma(P_1) \leq \gamma(P_2) \Rightarrow \text{Größe}(P_1) \leq \text{Größe}(P_2)$

$$\gamma(P1) + \gamma(P2) = \gamma(P3) \Rightarrow \text{Größe}(P1) + \text{Größe}(P2) = \text{Größe}(P3)$$

Beides ist mit der gegebenen Abbildungsvorschrift für γ der Fall.

2.4.2 Skalentypen

Abhängig von den Relationen, die es auf den Ausprägungen eines zu messenden Merkmals gibt, sind auf den Skalenwerten bestimmte Operationen möglich oder eben auch nicht möglich. Sind zum Beispiel die Merkmalswerte nicht additiv, so sind Additionen von Skalenwerten unzulässig, selbst wenn dies von der Art der Skala her möglich wäre. Abgestuft bezüglich der erlaubten Operationen werden fünf Skalentypen unterschieden (Tabelle 2.1).

TABELLE 2.1. Skalentypen

Skalentyp	Operationen	Beschreibung
Nominalskala	$= \neq$	Reine <i>Kategorisierung</i> von Werten. Die Skalenwerte sind weder vergleichbar noch miteinander verknüpfbar. Es ist nur nicht-parametrische Statistik möglich.
Ordinalskala	$= \neq < >$	Die Skalenwerte sind <i>geordnet</i> und miteinander vergleichbar. Die Bestimmung des Medianwerts ist möglich. Im übrigen kann nur nicht-parametrische Statistik betrieben werden. Mittelwerte oder Standardabweichungen können <i>nicht</i> berechnet werden.
Intervallskala	$= \neq < >$ Distanz	Werte sind geordnet; die <i>Distanz</i> zwischen je zwei Skalenwerten kann bestimmt werden. Der <i>Nullpunkt</i> der Skala ist <i>willkürlich</i> gewählt. Mittelwert und Standardabweichung sind bestimmbar.
Verhältnisskala (auch Rationalskala genannt)	$= \neq < >$ Distanz, (+ -), Vielfaches, %	Werte sind geordnet und in der Regel <i>additiv</i> ¹ . <i>Vielfache</i> und <i>Prozentwerte</i> sind bestimmbar. Die Skala hat einen <i>absoluten Nullpunkt</i> ; dieser repräsentiert das totale Fehlen der gemessenen Eigenschaft. Übliche parametrische Statistik ist möglich.
Absolutskala	$= \neq < >$ Distanz, (+ -), Vielfaches, %	Die Skalenwerte sind <i>absolute Größen</i> , das heißt, sie lassen sich in keine andere Skala umrechnen. Im Übrigen gleiche Eigenschaften wie Verhältnisskala.

BEISPIEL 2.2. Skalentypen

Nominalskala: Testergebnisskala mit den Werten {erfüllt, nicht erfüllt, nicht getestet}

Ordinalskala: Eignungsskala mit den Werten {- -, -, o, +, ++}

Intervallskala: Datumsskala für Zeit

Verhältnisskala: Anzahl-Codezeilen-Skala für Programmgröße

Absolutskala: Zählskala für die Anzahl gefundener Fehler in Programmen

2.4.3 Direkte und indirekte Maße

Es gibt einige interessierende Merkmale von Software und von Software-Entwicklungsprozessen, die sich in einfacher Weise direkt messen lassen. Solche Merkmale sind zum Beispiel Entwicklungskosten oder Durchlaufzeiten. Solche Maße nennen wir direkte Maße. Auch das in Beispiel 2.1 erwähnte Maß für die Größe von Programmen gehört zu den direkten Maßen.

Auf der anderen Seite gibt es Merkmale, für die es keine direkten Maße gibt oder wo das Messverfahren zu aufwendig wäre. In diesen Fällen behilft man sich, indem man *messbare Indikatoren* für das Merkmal bestimmt. Die Indikatoren sollen möglichst stark mit dem eigentlichen

¹ Die meisten in der Praxis vorkommenden Verhältnisskalen sind additiv. Additivität ist aber keine zwingende Eigenschaft.

zu messenden Merkmal korreliert sein. Die Indikatormasse bilden zusammen ein indirektes Maß für das interessierende Merkmal.

BEISPIEL 2.3. Messung der *Portabilität* (d.h. des Aufwands, ein System von einer bestehenden Hardware/Systemumgebung auf eine andere Hardware/Systemumgebung zu übertragen). Eine genaue Messung der Portabilität würde die Bestimmung des Quotienten $t_{\text{port}}/t_{\text{ent}}$ erfordern, wobei t_{port} der Aufwand für die Portierung und t_{ent} der Aufwand für die Neuentwicklung auf der neuen Hardware/Systemumgebung ist. Diese Messung kostet jedoch ein Vielfaches von dem, was sie nützt. Als Ausweg kann man die Messung von Portabilitäts-Indikatoren heranziehen (Bild 2.3).

Indikator	Skala	Messverfahren	Planwert	Schwellwert
Anzahl BS-Aufrufe/Anzahl Prozeduraufrufe	0-100%	Zählen im Code	5%	10%
Anteil Hardware- oder BS-abhängiger Module	0-100%	Zählen im Code	10%	15%
Anteil Nichtstandard Codezeilen	0-100%	Zählen, Vgl. mit ISO-Standard	2%	5%

BILD 2.3. Messung von Portabilität mit Indikatoren (BS=Betriebssystem)

2.5 Wichtige Maße für Ziele in der Software-Entwicklung

Zur Messung von *Produktzielen* interessiert man sich vor allem für Maße zur Messung der Funktionserfüllung (beispielsweise die Erfüllung von Leistungsanforderungen), sowie der Größe, der Zuverlässigkeit, der Benutzerfreundlichkeit oder der Pflfegbarkeit. Leider ist zu sagen, dass die meisten dieser Merkmale nur indirekt gemessen werden können und es bis heute keine allgemein anerkannten Sätze von Indikatoren für diese Merkmale gibt. Auch bei scheinbar einfachen Merkmalen wie der Produktgröße gehen die Meinungen auseinander, ob beispielsweise die Zahl der Codezeilen ein adäquates Größenmaß ist.

Im Bereich der Projektziele interessiert man sich insbesondere für Maße zur Messung von Aufwand, Durchlaufzeit (Dauer), Arbeitsfortschritt, Entwicklungskosten und Fehlerkosten. Im Gegensatz zu den Produktmerkmalen lassen sich die meisten Projektmerkmale verhältnismäßig einfach messen, wenn die notwendigen Basisdaten (zum Beispiel eine geeignete Zeitaufschreibung) vorliegen.

Aufgaben

- 2.1 Warum sind Zielsetzung und Zielverfolgung im Software Engineering besonders wichtig?
- 2.2 Geben Sie für die Skalentypen Nominalskala, Ordinalskala, Intervallskala, Verhältnisskala und Absolutskala je ein Beispiel aus dem täglichen Leben an.
- 2.3 In einem Unternehmen mit weltweiten Verkaufsniederlassungen sollen die Verkäuferinnen und Verkäufer durch ein zentrales Produktinformations- und Bestellsystem unterstützt werden. Dieses System soll
 - den Verkäuferinnen und Verkäufern aktuelle Produktinformationen liefern (insbesondere auch Angaben über lieferbare Mengen und Lieferfristen)
 - die Bestellungen der Verkäuferinnen und Verkäufer aufnehmen und an die Auftragsabwicklung weiterleiten
 - Auskunft über Status und Liefertermin von Bearbeitung befindlichen Aufträgen geben.

Die Verkäuferinnen und Verkäufer, welche dieses System benutzen sollen, verlangen in der Zeit zwischen 8.00 Uhr und 18.00 Uhr eine sehr hohe Systemverfügbarkeit.

Nehmen Sie an, Sie hätten den Auftrag, für diesen Benutzerwunsch ein messbares Ziel zu formulieren. Wie gehen Sie vor und wie sieht Ihre Zielformulierung aus?

- 2.4** Sie sind Leiterin bzw. Leiter eines Entwicklungsprojekts für ein neues Standardsoftwarepaket zur Erstellung von Steuererklärungen. In einem Strategiepapier des Produkt-Marketings ist für dieses Softwarepaket eine «besonders benutzerfreundliche Bedienung» verlangt. Was tun Sie?

- 2.5** Eine Firma vertreibt Systeme, mit denen technische Prozesse (z.B. Produktionsstrassen, Kraftwerke, Wasserversorgungssysteme) bedient und der Prozesszustand erfasst und angezeigt werden. Ein Kunde, welcher ein solches Leitsystem betreibt, bestellt zusätzlich ein Programm zur statistischen Auswertung von laufend erfassten Betriebsdaten. Nehmen Sie an, Sie sind verantwortlich für dieses Projekt. Sie haben kürzlich ein ähnliches System für einen anderen Kunden entwickelt und installiert. Der neue Auftrag unterscheidet sich von dem zuvor abgewickelten Auftrag wie folgt:

- ca. 50% höheres Datenvolumen
- zusätzliche Auswertungen in graphischer Form (das Vorgängersystem erzeugte nur Tabellen)
- andere Hardware, anderes Betriebssystem.

Weitere Betreiber interessieren sich für Ihr Auswertungssystem; Sie rechnen für die nächsten beiden Jahre mit mindestens fünf Aufträgen der gleichen Art. Sie wissen ferner, dass es in Ihrer Firma nur wenig Know-How über graphische statistische Auswertungen gibt. Ihre Abteilungsleiterin hat Sie beauftragt, einen Vorschlag für die Projektziele vorzulegen und sich zu überlegen, wie Sie die Zielverfolgung sicherstellen.

Welche Projektattribute und welche besonderen Qualitäten für das Produkt schlagen Sie aufgrund der spezifischen Projektsituation vor? Wie wollen Sie die Erreichung der vorgeschlagenen Ziele kontrollieren?

Ergänzende und vertiefende Literatur

Briand, Morasca und Basili (1996) beschreiben grundlegende Eigenschaften wichtiger Produktmaße. In einem Vergleich mit anderen Ansätzen erschließt dieses Papier ferner weitere Literatur zur Fundierung des Messens im Software Engineering.

Fenton (1991) sowie Conte, Dunsmore und Shen (1986) sind Monographien über Messen im Software Engineering. Das Buch von Conte, Dunsmore und Shen ist leider vergriffen und nur noch über Bibliotheken zugänglich.

Zitierte Literatur

Briand, L.C., S. Morasca and V.R. Basili (1996). Property-Based Software Engineering Measurement. *IEEE Transactions on Software Engineering* **22**(1). 68-85.

Conte, S.D., H.E. Dunsmore, V.Y. Shen (1986). *Software Engineering Metrics and Models*. Menlo Park, CA., etc.: Benjamin/Cummings.

Fenton, N.E. (1991). *Software Metrics. A Rigorous Approach*. London, etc.: Chapman&Hall.

Weinberg G., E. Schulman (1974). Goals and Performance in Computer Programming. *Human Factors* **16**, 1. 70-77.

3. Der Software-Prozess

Software-Prozesse beschreiben den Ablauf der Entwicklung und der Pflege von Software. In diesem Kapitel werden zunächst einige grundlegenden Phänomene im Zusammenhang mit Software-Prozessen beschrieben. Dann werden ausgewählte Prozessmodelle für die Entwicklung von Software vorgestellt. Anschließend wird die Rolle von Prototypen und von Wiederverwendung und Beschaffung im Entwicklungsprozess diskutiert. Ausführungen zum Prozess der Software-Pflege schließen das Kapitel ab.

3.1 Grundlagen

3.1.1 Software-Prozesse

Den Ablauf eines Vorhabens mit der Beschreibung der Schritte, der beteiligten Personen, der für diesen Ablauf benötigten Informationen und der dabei entstehenden Informationen nennen wir einen *Prozess*.

DEFINITION 3.1. *Prozess.* Eine Folge von Schritten, die zur Erreichung eines gegebenen Zwecks ausgeführt wird. (IEEE 610.12).

Wir unterscheiden dabei zwischen ad-hoc-Prozessen, die spontan, individuell und unregelmäßig ablaufen und systematischen Prozessen, deren Ablauf geplant und gelenkt wird. Auch die *Herstellung und Pflege von Software* sind Prozesse.

Im nicht professionellen Bereich wird Software typischerweise nach einem *ad-hoc-Prozess* entwickelt und gepflegt. Ein solcher Prozess hat folgende charakteristischen Merkmale: Es gibt in der Regel nur vage Sachziele und keine Termin- und Kostenvorgaben. Spezifikationen werden keine, Entwürfe nicht oder nur bruchstückhaft gemacht. Getestet wird nicht oder nur ansatzweise; Qualität ist kein Thema. Die Entwicklung hat oft den Charakter des Probierens. Die Produkte sind eher klein, kurzlebig und nicht dokumentiert. In der Regel handelt es sich um Einpersonalarbeit für den Eigengebrauch. Pflegemaßnahmen erfolgen spontan immer dann, wenn mit der Software Probleme auftreten, die sich nicht anders lösen lassen.

Im professionellen Bereich hat diese Art von Arbeit ausschließlich bei experimentellen Entwicklungen im Forschungsbereich sowie bei der Erstellung von Wegwerf-Prototypen (vgl. Kapitel 3.3) ihre Berechtigung. Man findet sie jedoch leider häufig auch in der sogenannten Endbenutzer-Entwicklung, wo Leute für ihren PC Software in dieser Art zusammenbasteln.

Wird Software als Produkt oder als Teil eines Produkts entwickelt, so braucht es zwingend *systematische* (d.h. geplante und gelenkte) *Prozesse*, sofern man keine Software-Katastrophen will. Die Abwicklung wird geplant; es bestehen klare Termin-, Kosten- und Sachziele. Der Ablauf des Prozesses wird überprüft. Mit Lenkungsmaßnahmen werden Störungen im Ablauf sowohl präventiv als auch reaktiv bekämpft. Qualität ist wichtig, da das fertige Produkt anschließend gepflegt und in der Regel weiterentwickelt werden muss.

Systematische Pflege (Wartung) ist ein kontinuierlicher Prozess, welcher die Erhaltung der Gebrauchstauglichkeit eines Software-Produkts zum Ziel hat. Der Prozess regelt nicht nur die Reaktion auf aufgetretene Probleme, sondern plant und lenkt auch die Weiterentwicklung der Software und ihre laufende Anpassung an ein sich wandelndes Umfeld.

3.1.2 Software-Projekte

Jede systematische Entwicklung von Software sowie alle größeren, abgrenzbaren Pflegevorhaben benötigen einen Abwicklungsrahmen. Dies ist in der Regel ein Software-Projekt.

DEFINITION 3.2. *Projekt.* Eine zeitlich befristete Organisationsform zur Bearbeitung einer gegebenen Aufgabe. Dient zur Regelung der Verantwortlichkeiten und zur Zuteilung der für die Arbeit notwendigen Ressourcen.

Software-Projekte unterscheiden sich von klassischen technischen Entwicklungsprojekten. Der wesentlichste Unterschied ist, dass in den klassischen Disziplinen das entstehende materielle Produkt eine Reihe von natürlichen Ansatzpunkten für Fortschrittskontrolle und Problemerkennung bietet, die es bei Software als einem immateriellen Produkt nicht gibt.

Software-Projekte müssen daher sorgfältiger geplant, überprüft und gelenkt werden als andere technische Projekte. Mögliche Mittel hierfür sind die Verwendung (und Einhaltung) geeigneter Prozesse sowie ein adäquater Umgang mit den Projektrisiken.

FESTSTELLUNG 3.1. Jede systematische Entwicklung von Software und jedes größere Software-Pflegevorhaben wird als Projekt organisiert.

3.1.3 Der Software-Lebenslauf

Der Software-Lebenslauf ist ein grundlegendes Phänomen für das Verständnis von Software-Prozessen. Betrachtet man eine beliebige Software-Komponente isoliert für sich, so stellt man fest, dass jede solche Komponente einen *Lebenslauf* hat, welcher die Stadien Initiierung – Entwicklung – Nutzung durchläuft.

In der Entwicklung laufen für die Erstellung einer solchen Einzelkomponente folgende Tätigkeiten ab:

- Analysieren der Aufgabe und *Spezifizieren der Anforderungen*; Dokumentieren und Prüfen der Anforderungen
- *Konzipieren der Lösung* (Grobkonzept, Architekturentwurf); Dokumentieren und Prüfen des Konzepts
- *Entwerfen der Lösung im Detail*; Dokumentieren und Prüfen des Entwurfs
- *Codieren* des Programmstücks für die betreffende Einzelkomponente; *Testen* des Programmstücks
- *Integrieren* mit den Programmstücken anderer Einzelkomponenten; Dokumentieren und *Testen* der Integrationsschritte
- *Installieren und Testen* der Komponente als Teil des Gesamtsystems.

DEFINITION 3.3. *Software-Lebenslauf (software life cycle).* Zeitraum, in dem eine Software-Komponente bearbeitet oder benutzt wird. Beginnt mit der Initiierung der Komponente und endet mit ihrer endgültigen Außerbetriebsetzung. Umfasst typisch die Stadien Initiierung, Entwicklung und Nutzung, wobei die Entwicklung wiederum in die Stadien Spezifikation der Anforderungen, Konzept der Lösung, Entwurf und Programmierung, Zusammensetzung mit anderen Komponenten und Inbetriebnahme (einschließlich der Überprüfung des Entwickelten nach jedem Schritt) zerfällt.

FESTSTELLUNG 3.2. Je kleiner ein Stück Software ist, desto mehr verläuft ihr Lebenslauf linear. Bei größeren Komponenten und bei ganzen Systemen gibt es dagegen Iterationen in allen Stadien des Lebenslaufs.

Solche Iterationen werden beispielsweise durch sich ändernde Anforderungen oder durch die Entdeckung und Korrektur von Fehlern verursacht. Außerdem gibt es auch Prozessmodelle, die

von vornherein Iterationen vorsehen (vgl. die Ausführungen zu Wachstumsmodellen in Abschnitt 3.2).

Während der Nutzung eines Systems löst jede Anpassung oder Erweiterung die gleiche Folge von Entwicklungsaktivitäten für die anzupassenden oder neuen Komponenten aus.

3.1.4 Software-Evolution

Lehman teilt in seinem fundamentalen Artikel von 1980 die Software in drei Klassen ein.

- Software vom *S-Typ* ist solche, die vollständig durch eine formale Spezifikation beschrieben werden kann. Die Entwicklung ist erfolgreich, wenn die resultierenden Programme nachweislich die Spezifikation erfüllen. Typische Beispiele für S-Software sind Programme zum Sortieren von Daten oder zur Berechnung mathematischer Funktionen.
- Software vom *P-Typ* ist solche, die ein spezifisches, abgegrenztes Problem löst. Die Entwicklung ist erfolgreich, wenn das Problem zufriedenstellend gelöst ist. Typische Beispiele für P-Software sind Programme, welche gegebene Modelle, etwa Festigkeitsmodelle in der Statik oder Wettervorhersagemodelle in der Meteorologie berechnen.
- Software vom *E-Typ* ist solche, welche eine in der realen Welt eingebettete Anwendung realisiert. Die Entwicklung ist erfolgreich, wenn die Anwender mit der Software zufrieden sind.

FESTSTELLUNG 3.3 (LEHMAN). Nur Software vom S-Typ ist stabil. Software vom P- und E-Typ ist einer *Evolution* unterworfen. Software vom E-Typ trägt selbst zur Evolution bei.

Evolution bedeutet in diesem Zusammenhang, dass ein Produkt aufgrund des sich wandelnden Umfelds selbst einem permanenten Wandel unterliegt. Software vom E-Typ ist selbst Teil dieses Umfelds und trägt mit zum Wandel bei: Ein Produkt löst durch seine Benutzung Veränderungen in seiner Umwelt aus, die ihrerseits Veränderungen des betreffenden Produkts erforderlich machen.

Die gleichen Mechanismen der Evolution finden wir auch bei allen anderen technischen Produkten. Bei Software geht die Evolution jedoch besonders schnell. Dies hat zwei wichtige Konsequenzen:

FESTSTELLUNG 3.4. Es ist unmöglich, Software vom P- oder E-Typ so zu entwickeln, dass sie während einer mehrjährigen Lebensdauer ohne jegliche Änderungen ihre Aufgaben zur vollen Zufriedenheit der Benutzer erfüllt.

Pflege (Wartung) ist bei dieser Art von Software kein Unfall, der durch sorgfältigere Entwicklung hätte vermieden werden können, sondern ein unvermeidlicher Bestandteil der zu leistenden Arbeit. Bis zu einem Drittel des Gesamtaufwands für ein Software-Produkt entfällt auf die Weiterentwicklung des Produkts während der Nutzung (vgl. Bild 1.3). Alle Überlegungen zu Software, seien sie technischer oder wirtschaftlicher Art, dürfen daher nicht isoliert nur für die Software-Entwicklung gemacht werden, sondern müssen sich auf die gesamte Lebensdauer der Software beziehen.

FESTSTELLUNG 3.5. Dauert die Entwicklung von Software vom P- oder E-Typ länger als ca. ein halbes Jahr, so ist es kaum möglich, nach einem Satz einmal erhobener und dann festgeschriebener Anforderungen so zu entwickeln, dass das fertige Produkt bei seiner Inbetriebnahme die Bedürfnisse der Leute, die es benutzen sollen, optimal befriedigt.

Sich verändernde Anforderungen sind bei P- und E-Software demnach ebenfalls kein Unfall, sondern problembedingt. Da solche Veränderungen Instabilitäten und Mehrkosten in den Projekten zur Folge haben können, muss dieses Phänomen im gewählten Software-Prozessmodell möglichst berücksichtigt werden.

3.1.5 Meilensteine

Meilensteine sind das wichtigste Mittel, um den Ablauf eines Prozesses zu strukturieren und den Fortschritt wirksam zu kontrollieren.

DEFINITION 3.4. Ein *Meilenstein* ist eine Stelle in einem Prozess, an dem ein geplantes Ergebnis vorliegt.

Ein Meilenstein wird *geplant*, indem das zu erreichende Ergebnis und die dafür zur Verfügung stehenden Ressourcen (Zeit, Geld, Personal, etc.) festgelegt werden. Ein Meilenstein ist *erreicht*, wenn das verlangte Ergebnis nachweislich vorliegt. Durch Vergleichen des geplanten mit dem effektiven Ressourcenverbrauch ist eine wirksame, quantitative *Fortschrittskontrolle* möglich.

Meilensteine sind nur wirksam, wenn das mit einem Meilenstein verknüpfte Ergebnis so beschrieben ist, dass die Erreichung des Meilensteins *gemessen* oder mit hinreichender Sicherheit *überprüft* werden kann. Bild 3.1 zeigt drei Beispiele.

Ergebnis in einem Software-Prozess	Eignung für Meilenstein
Codierung zu 90% beendet	unbrauchbar, da nicht messbar, sondern nur schätzbar
Die Komponente xyz hat internen Abnahmetest bestanden	geeignet, da messbar (an Hand des Abnahmetest-Protokolls)
Die Anforderungsspezifikation liegt vor	geeignet, wenn Inhalt und Form des Dokuments durch ein Review überprüft werden

BILD 3.1. Eignung von Zwischenresultaten für Meilensteine

3.2 Prozessmodelle für die Entwicklung von Software

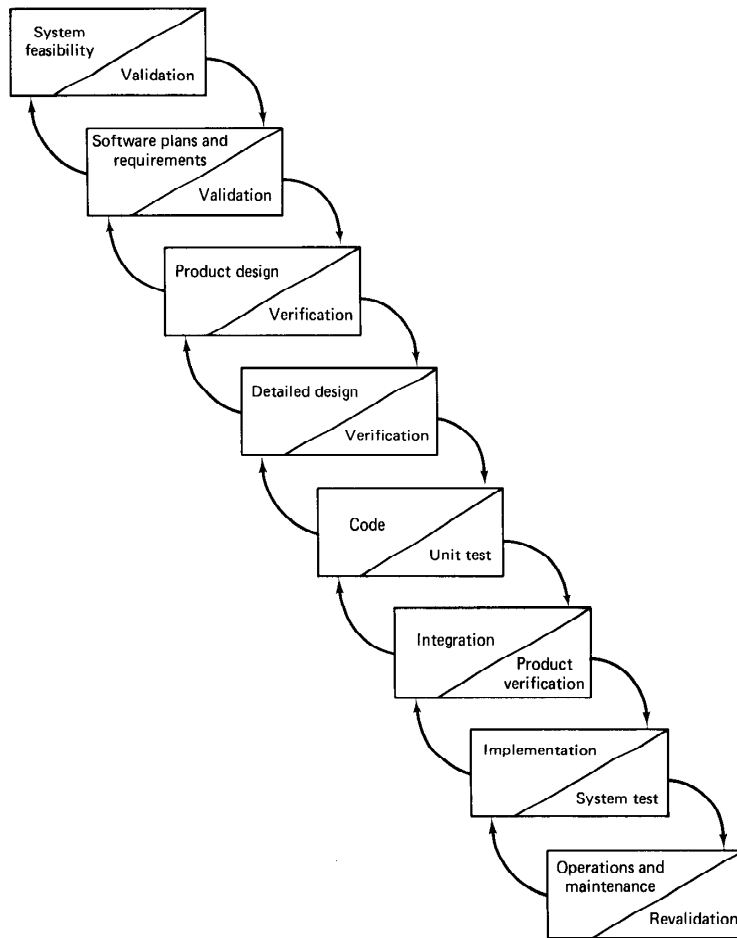
Da man Software nicht sehen kann, ist auch der Ablauf, in welchem sie entsteht, ohne besondere Maßnahmen nicht erkennbar. Es ist daher sehr wichtig, eine *Modellvorstellung* über den Ablauf der Entstehung von Software zu haben und sowohl die Planung als auch die Kontrolle der Entwicklung an diesem Modell zu orientieren. Nur so ist ein Software-Projekt führbar.

Ein Software-Prozessmodell ist genau eine solche Modellvorstellung. Die Wahl eines geeigneten Prozessmodells ist daher eine der Grundlagen für eine erfolgreiche Software-Projektführung. In den folgenden Abschnitten werden ausgewählte Prozessmodelle vorgestellt und charakterisiert.

3.2.1 Das Wasserfall-Modell

Das sogenannte Wasserfall-Modell wurde von Royce (1970) entwickelt und später vor allem von Boehm (1981) propagiert. Es ist das älteste systematische Prozessmodell für die Entwicklung von Software.

Das Wasserfall-Modell überträgt den Software-Lebenslauf, d.h. die bei der Entwicklung von Einzelkomponenten beobachtete Reihenfolge von Tätigkeiten, auf die Entwicklung eines ganzen Systems (Bild 3.2). Jede Tätigkeit bildet eine *Entwicklungsphase*. Am Ende jeder Phase steht ein Prüfschritt, welcher sicherstellen soll, dass eine Phase erst verlassen wird, wenn alle benötigten Phasenergebnisse in geprüfter und akzeptierter Form vorliegen. Trotzdem geht das Wasserfall-Modell davon aus, dass in einer Phase Probleme in den Vorgaben, d.h. den Ergebnissen der vorangegangenen Phase, auftreten können. Es sind daher lokale Iterationen zwischen aufeinanderfolgenden Phasen vorgesehen, um solche Probleme zu beheben.



(Quelle: Boehm, 1981)

BILD 3.2. Das Wasserfall-Modell

DEFINITION 3.5. Wasserfall-Modell. Ein Modell für den Software-Entwicklungsprozess, welches die Entwicklung in eine Sequenz von Entwicklungs- und Prüfaktivitäten unterteilt. Die Reihenfolge der Entwicklungsaktivitäten folgt dem Software-Lebenslauf.

Es gibt eine große Zahl von Varianten des ursprünglichen Wasserfall-Modells, welche sich meist geringfügig in Benennung und Reihenfolge der Phasen unterscheiden. Allen diesen Modellen ist jedoch gemeinsam, dass die Unterteilung der Entwicklung in Phasen nach einer linearen Reihenfolge von Tätigkeiten erfolgt.

Aufgrund der Software-Evolution und aufgrund der Tatsache, dass die erkannten Fehler nicht immer nur in der gegenwärtigen und der vorangegangenen Phase gemacht wurden, sind bei der Anwendung des Wasserfall-Modells häufig Iterationen über mehrere Phasen hinweg erforderlich (Bild 3.3). Dies erschwert die Projektführung und hat dem Wasserfall-Modell sehr viel (berechtigte) Kritik eingetragen. Trotz seiner unbestreitbaren Vorteile (Einfachheit, Modellierung des natürlichen Lebenslaufs) und seiner immer noch beachtlichen Verbreitung sollte das Wasserfall-Modell in der Interpretation einer Folge von Entwicklungstätigkeiten heute nicht mehr verwendet werden. Gleiches gilt für alle ähnlichen Modelle, welche die Phasen als Tätigkeiten auffassen.

3.2.2 Ergebnisorientierte Phasenmodelle

Ergebnisorientierte Phasenmodelle machen wie das Wasserfall-Modell den Software-Lebenslauf zur Grundlage des Prozesses. Sie gehen also ebenfalls davon aus, dass zunächst alle Anforderungen spezifiziert werden, dann das System vollständig entworfen wird, dann codiert und ge-

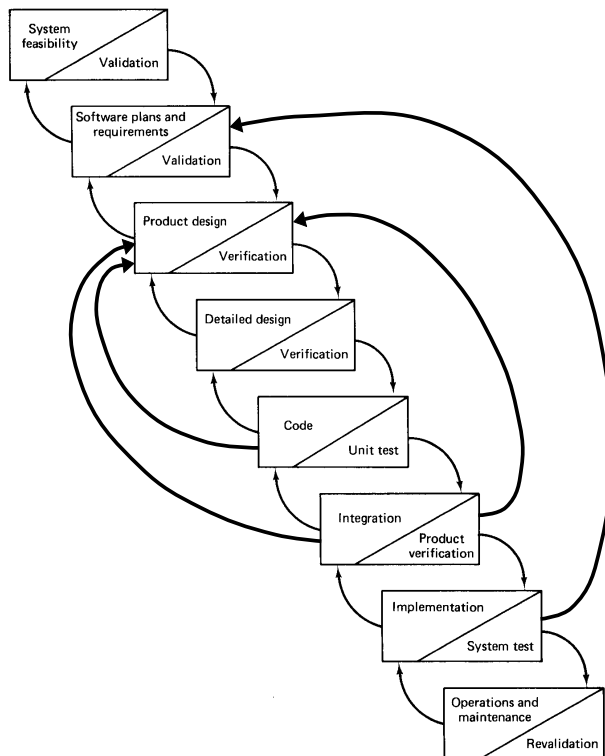


BILD 3.3. Mehrfache Iterationen im Projektablauf mit dem Wasserfall-Modell

testet wird, usw. In einem ergebnisorientierten Phasenmodell ist eine Phase jedoch keine Tätigkeit wie beim Wasserfall-Modell, sondern ein *Zeitintervall*.

DEFINITION 3.6. Ergebnisorientiertes Phasenmodell. Ein Modell für den Software-Entwicklungsprozess, welches die Entwicklung in eine Sequenz aufeinanderfolgender Zeitabschnitte (Phasen) unterteilt und in jeder Phase einen Teil der insgesamt zu liefernden Ergebnisse erarbeitet. Die Reihenfolge der Phasen orientiert sich am Software-Lebenslauf.

Die Phasen in ergebnisorientierten Phasenmodellen sind nach diesem Ergebnis oder nach der vorherrschenden Tätigkeit, die zum Ergebnis führt, benannt. Es gibt keine Phasen-Iteration. In jeder Phase können (bzw. müssen) notwendige Nacharbeiten an in früheren Phasen erzielten Ergebnissen durchgeführt werden.

Jeder Phasenabschluss bildet einen *Meilenstein* in der Entwicklung. Weitere Meilensteine können bei Bedarf hinzugefügt werden. Die geforderten Ergebnisse der frühen Phasen sind typisch Dokumente; später bestehen die Ergebnisse aus Code und Dokumenten.

Wie beim Wasserfall-Modell gibt es auch hier eine Vielzahl von Varianten, die sich in Reihenfolge und Benennung der Phasen unterscheiden. Bild 3.4 zeigt einen typischen Vertreter.

Ergebnisorientierte Phasenmodelle haben folgende Vorteile:

- Sie sind leicht verständlich und folgen dem natürlichen Software-Lebenslauf.
- Sie sind ein geeignetes Instrument für die Projektführung.
- Sie fördern planvolles Vorgehen mit sorgfältigem Spezifizieren, Konzipieren, Prüfen und Dokumentieren. Sie verhindern wildes Drauflos-Programmieren nach dem Motto: Einhacken–Probieren–Absturz–Ändern–Probieren–Absturz... .
- Ausführliche Spezifikationen und Entwürfe tragen erheblich dazu bei, Fehler früh zu erkennen und damit Zeit und Kosten zu sparen.

Diesen Vorteilen stehen folgende Nachteile gegenüber:

- Die Grundannahme, dass ganze Systeme genauso wie Einzelkomponenten planvoll in einer Abfolge von Schritten *konstruiert* werden können, ist vor allem für große Systeme mit langer Entwicklungs- und Lebensdauer meistens *falsch*. Solche Systeme sind in der Regel nicht konstruierbar, sondern sie *wachsen* evolutionär. Dies gilt insbesondere dann, wenn die Anforderungen an ein System nicht a priori genau bekannt sind.
- Lauffähige Systemteile entstehen erst relativ spät:
 - Die lange Durststrecke, in der nur Dokumentation produziert wird, ist nachteilig für die Motivation der Projektbeteiligten.
 - Unter Umständen wird erst sehr spät erkannt, dass vorgesehene Lösungen technisch nicht realisierbar sind, das System nicht die geforderten Leistungen erbringt oder dass es nicht das leistet, was der Auftraggeber eigentlich wollte.
- Das System muss in einem Schritt komplett in Betrieb genommen werden.

Ergebnisorientierte Phasenmodelle eignen sich daher vor allem dann, wenn

- das Projekt nicht allzu groß ist
- die Aufgaben, welche das zu erstellende System erfüllen soll, genau bekannt und präzise beschreibbar sind
- die Beteiligten über Know How im betreffenden Problembereich verfügen (z.B. ähnliche Projekte bereits abgewickelt haben)
- das Entwicklungsrisiko eher gering ist.

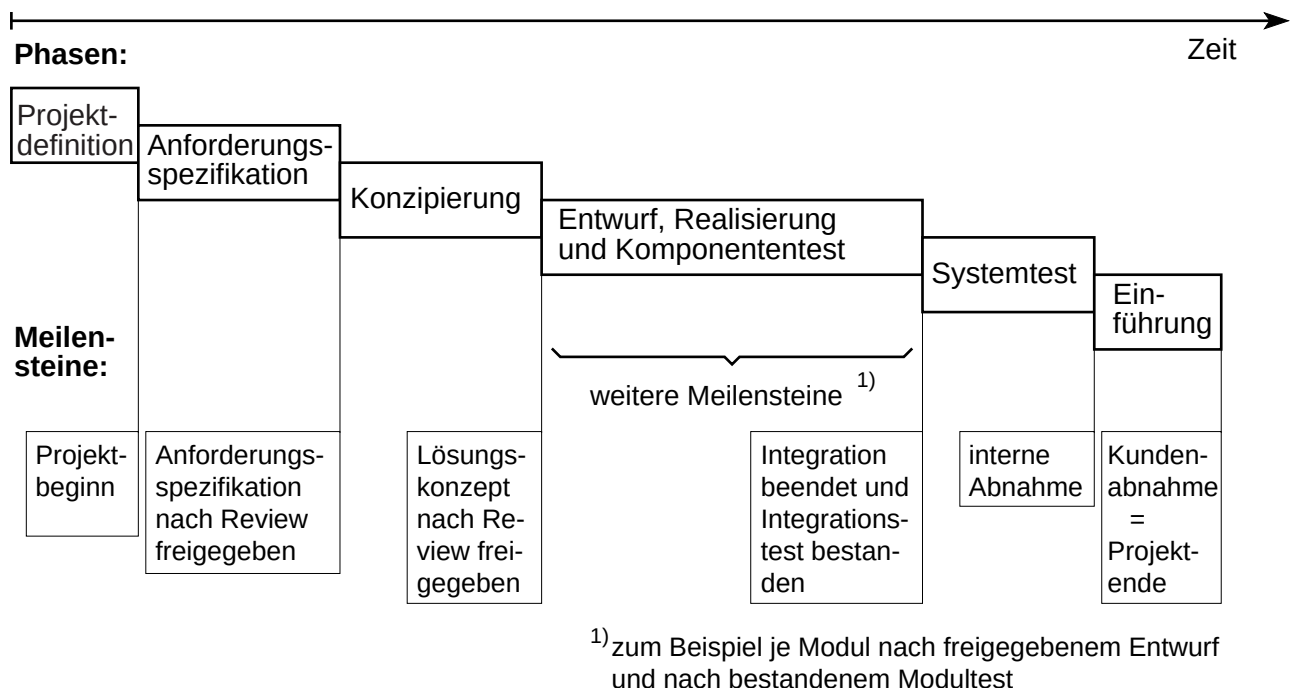


BILD 3.4. Beispiel eines ergebnisorientierten Phasenmodells

3.2.3 Wachstumsmodelle

Wachstumsmodelle stellen den Gedanken der *Software-Evolution* in den Mittelpunkt. Ein System wird nicht konstruiert, sondern es *wächst* in einer Reihe aufeinanderfolgender Schritte.

Es gibt verschiedene Varianten von Wachstumsmodellen unter verschiedenen Namen, z.B. Versionen-Modell, evolutionäres Modell, inkrementelles Modell. Die Ideen zu dieser Art von Modellen gehen auf Basili und Turner (1975) zurück. Sie wurden von IBM für die Entwicklung

sehr großer technischer Software aufgenommen (Mills et al., 1980) und später vor allem von Gilb (1988) verbreitet.

Der Grundgedanke ist immer der folgende: Ausgehend von einer groben Zusammenstellung der Gesamtfunktionalität des gewünschten Systems wird eine Folge von *Lieferungen* mit Lieferumfang und Liefertermin definiert. Jede Lieferung ist *betriebsfähig* und kann nach Auslieferung sofort in Betrieb genommen werden. Erfahrungen mit frühen Lieferungen können bei späteren Lieferungen berücksichtigt werden.

DEFINITION 3.7. Wachstumsmodell. Ein Modell für den Software-Entwicklungsprozess, welches die Entwicklung in eine Folge von Iterationen unterteilt. In jeder Iteration wird ein vollständiges Teilergebnis mit betriebsfähiger Software erarbeitet und ausgeliefert.

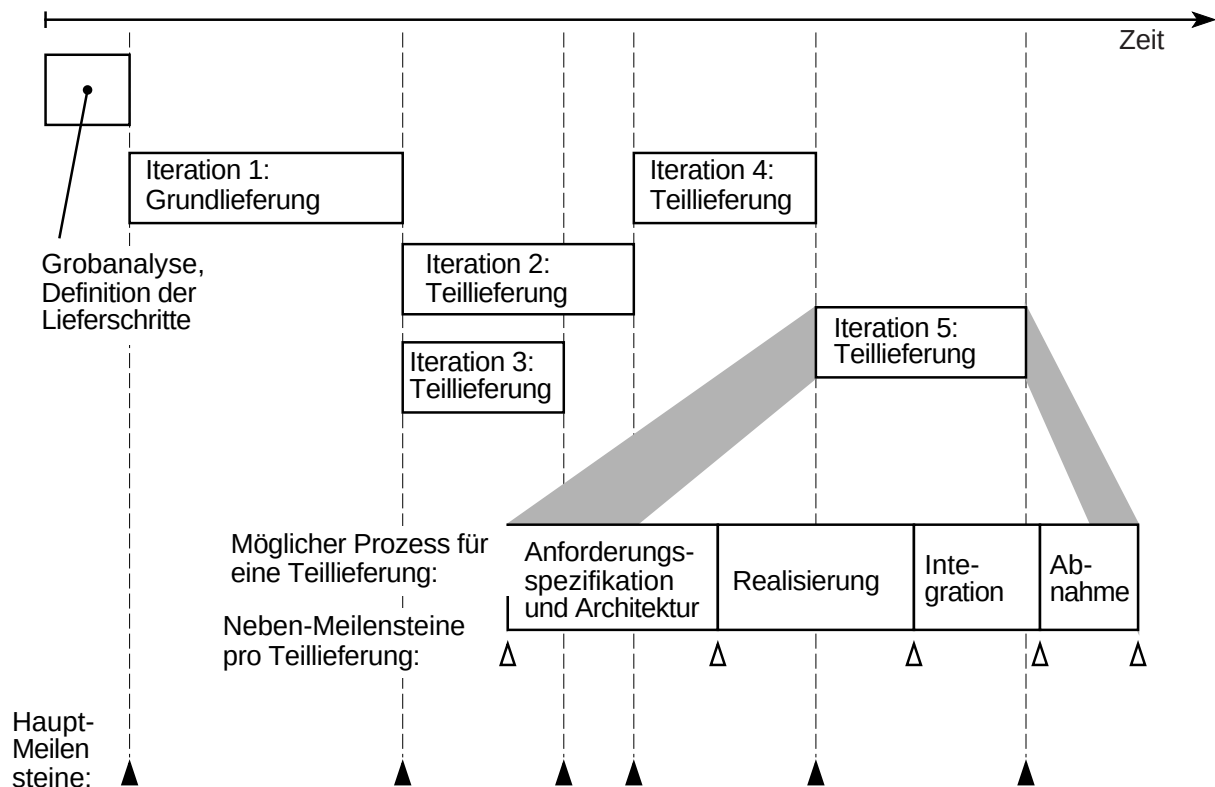


BILD 3.5. Beispiel eines Projektablaufs mit einem Wachstumsmodell

Jede Lieferung wird als weitgehend autonomes Teilprojekt organisiert. Da diese Teilprojekte in der Regel klein und kurz sind, kann ein ergebnisorientiertes Phasenmodell als Prozessmodell verwendet werden. Bild 3.5 zeigt den zeitlichen Ablauf eines nach einem Wachstumsmodell organisierten Projekts.

Bei der Definition des Umfangs und der Reihenfolge der Lieferungen wird immer ein schrittweiser Ausbau des Systems bis zum gewünschten Endzustand geplant. Dabei können verschiedene Merkmale schrittweise ausgebaut werden, zum Beispiel die Funktionalität, der Bedienungskomfort, die Leistung oder die Verwendbarkeit. Mit letzterem ist beispielsweise der Ausbau von einer Einzweck- zu einer Vielzweckanwendung oder von einem dedizierten zu einem portablen System gemeint.

In Fällen, wo ein Auftraggeber keine Teillieferungen will, bzw. man nicht mit Teilprodukten auf den Markt gehen will, kann ein Wachstumsmodell trotzdem sinnvoll eingesetzt werden. Die Lieferungen erfolgen dann intern. Vorteile sind eine bessere Projektkontrolle und lieferfähige Teile, wenn der Liefertermin für das Gesamtsystem nicht eingehalten werden kann.

Wachstumsmodelle haben die folgenden Vorteile:

- Sie modellieren das natürliche Verhalten der meisten großen Systeme.
- Sie sind ein sehr geeignetes Instrument zur Projektführung: Die Projektführung im Großen erfolgt mit den Meilensteinen der Lieferungen. Da jede Lieferung lauffähig sein muss, ist eine wirksame Kontrolle der Meilensteine möglich. Im Kleinen wird jede Lieferung über die Meilensteine eines ergebnisorientierten Phasenmodells kontrolliert.
- Es entstehen sehr schnell lauffähige Teile des Systems. Dies fördert die Motivation der Projektbeteiligten und hilft dem Auftraggeber, seine größten Nöte frühzeitig zu lindern.
- Das System kann sanft eingeführt werden. Es können Erfahrungen in realen Pilotanwendungen gesammelt werden. Umstellung, Schulung der Mitarbeiter, etc. können schrittweise erfolgen.

Dem stehen folgende Nachteile gegenüber:

- Es besteht die Gefahr, dass anstelle eines geschlossenen Systems viele nicht ganz zusammenpassende Teilsysteme entstehen.
- Die Konzepte und Strukturen der ersten Lieferungen können durch Ergänzungen und Änderungen in späteren Lieferungen bis zur Unkenntlichkeit entstellt werden, so dass sich die Pfllegbarkeit des Gesamtsystems drastisch verschlechtert.

Systematisches, sorgfältig geplantes Arbeiten ist daher beim Einsatz von Wachstumsmodellen besonders wichtig. Die Verwendung eines Wachstumsmodells darf kein Vorwand für ein Abgleiten in die ad-hoc-Entwicklung sein!

Wachstumsmodelle eignen sich vor allem, wenn

- das System groß ist und eine lange Entwicklungszeit hat
- nicht die volle Funktionalität sofort erforderlich ist
- man ein Software-Vorhaben nicht vorab in allen Einzelheiten definieren kann, sondern zunächst Erfahrungen sammeln will oder muss
- ein Basisprodukt schnell auf dem Markt bzw. beim Kunden sein soll.

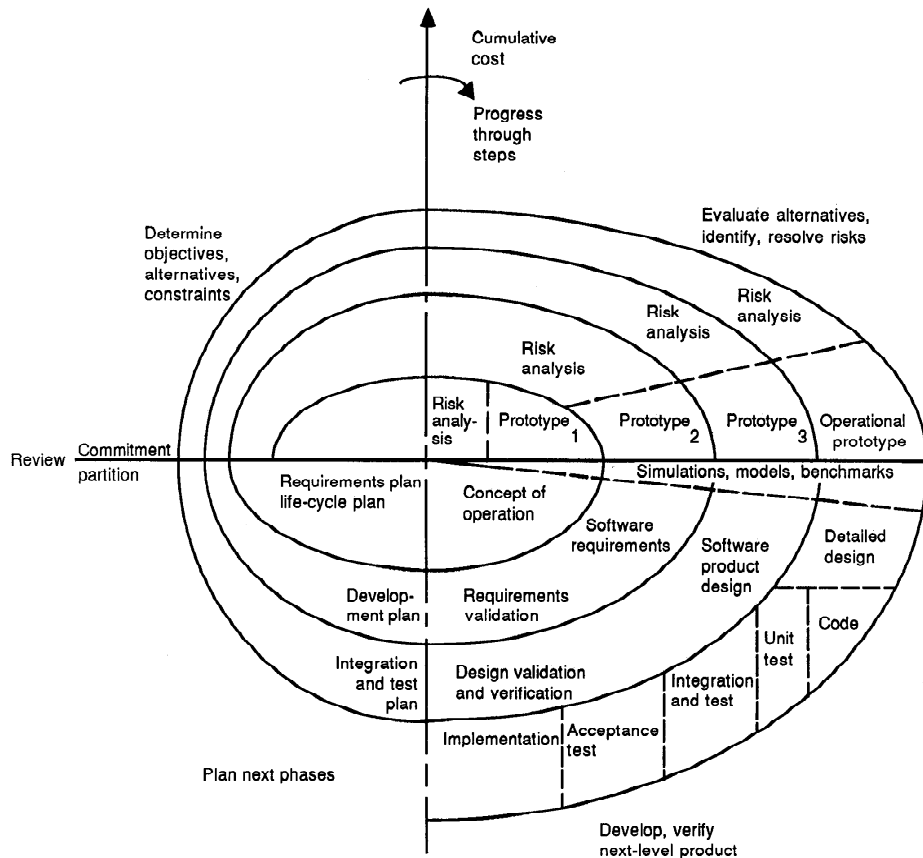
3.2.4 Das Spiralmodell

Das Spiralmodell (Bild 3.6) ist eine von Boehm (1988) stammende Weiterentwicklung des Wasserfall-Modells. Es ist vor allem für die Entwicklung *großer, risikoreicher* Systeme gedacht. Anstelle der zyklischen Abfolge "Entwickeln, prüfen, entwickeln, prüfen..." tritt beim Spiralmodell ein vierteiliger Zyklus, der in der graphischen Modell-Darstellung den vier Quadranten des Koordinatensystems entspricht:

1. Planung des nächsten Spiralumlaufs
2. Zielsetzung bzw. Zielkorrektur
3. Untersuchung von Varianten und ihren Risiken, Variantenentscheid
4. Entwicklung und Prüfung (dieser Schritt korrespondiert mit dem Wasserfall-Modell).

Der Zyklus wird viermal durchlaufen, nämlich für *Systemdefinition*, *Anforderungen* an die Software, *Konzipierung* (Architektur-Entwurf) und *Realisierung* (Detailentwurf, Codierung, Test, Integration und Installation). Wichtig ist vor allem der dritte Schritt im Zyklus, wo die *Risiken* der Entwicklung erkannt und kontrolliert werden. Boehm selbst bezeichnet das Spiralmodell als *risikogesteuert*. Das Hauptmittel hierfür sind *Prototypen*.

Gelegentlich findet man unter dem Namen „Spiralmodell“ auch Modelle, die eigentlich Wachstumsmodelle sind, bei denen aber jeder Wachstumsschritt als ein Spiralumlauf dargestellt ist.



(Quelle: Boehm, 1988)

BILD 3.6. Das Spiralmodell

3.3 Prototypen

Prototypen sind ein zentrales Mittel zur frühzeitigen Erkennung und Lösung von Problemen in der Software-Entwicklung.

DEFINITION 3.8. Prototyp. Ein lauffähiges Stück Software, welches kritische Teile eines zu entwickelnden Systems vorab realisiert.

Der Prozess der Erstellung und Nutzung von Prototypen wird als *Prototyping* bezeichnet. Es werden drei Arten von Prototyping unterschieden: *exploratives*, *experimentelles* und *evolutionäres* Prototyping (Floyd, 1984; Kieback, Lichter, Schneider-Hufschmidt und Züllighoven 1992).

Exploratives Prototyping dient zur Klärung und Bestimmung von Anforderungen an Hand von lauffähigen Programmen, welche den zu klärenden Ausschnitt aus dem Gesamtproblem realisieren. Exploratives Prototyping wird einerseits verwendet, um mit sogenannten *Demonstrationsprototypen* die prinzipielle Nützlichkeit und Machbarkeit von Systemideen zu demonstrieren (v.a. im Bereich der Akquisition und der Projekt-Initiierung). Andererseits dient es dazu, ein konkretes, von den zukünftigen Benutzern erprobbares und kritisierbares Modell einer geplanten Informatik-Lösung zu schaffen. An Hand eines solchen *Prototyps im engeren Sinn* können die Angemessenheit von Anforderungen und die Tauglichkeit von Lösungen für den geplanten Verwendungszweck untersucht werden. Beispielsweise können so Benutzerbedürfnisse geklärt oder geplante Benutzerschnittstellen erprobt werden. Bei Produktentwicklungen kann exploratives Prototyping außerdem zur Gewinnung von Wissen über einen bisher nicht beherrschten Problembereich dienen.

Experimentelles Prototyping wird verwendet, um die Realisierbarkeit kritischer Systemteile zu untersuchen und um Entwurfsalternativen zu bewerten. Beispielsweise kann so geklärt werden, ob bestimmte geforderte Leistungen (Reaktionszeiten, Datenraten, ...) erbracht werden können. Solche Prototypen werden auch als *Labormuster* bezeichnet.

Demonstrationsprototypen, Prototypen im engeren Sinn und Labormuster sind *Wegwerf-Prototypen*. Das geplante System wird auf der Grundlage des mit den Prototypen erworbenen Wissens neu entwickelt. Es ist zulässig, solche Wegwerf-Prototypen nicht zu dokumentieren und schnelle, softwaretechnisch unsaubere Lösungen zu verwenden. Es muss allerdings sichergestellt sein, dass niemand der Versuchung erliegt, aus Software, die als Wegwerfprototyp geschrieben wurde, dann doch durch Weiterentwicklung ein Produkt zu machen. Ist ein solcher Ausbau beabsichtigt, so muss ein evolutionäres Prototyping gewählt werden (siehe unten). Die Erstellung von Wegwerfprototypen ist wirtschaftlich, wenn es gelingt, bei der eigentlichen Systementwicklung aufgrund der Erfahrung mit den Prototypen mehr Kosten zu sparen, als die Prototypen selbst gekostet haben, oder wenn mit Prototypen ein gefährliches Risiko wesentlich gemindert werden kann (vgl. Kapitel 4, Risiken).

Im Gegensatz dazu ist beim *evolutionären Prototyping* der Prototyp ein *Pilotsystem*, das den Kern des zu entwickelnden Systems bildet, operational eingesetzt wird und durch schrittweise Ergänzungen und Erweiterungen zum geplanten System weiterentwickelt wird. Dementsprechend muss ein evolutionärer Prototyp sorgfältig und nach allen Regeln der Kunst entwickelt werden. Evolutionäres Prototyping entspricht im Vorgehen einem Wachstumsmodell.

Das Arbeiten mit Prototypen ist somit kein eigenständiges Modell für die Entwicklung von Software:

- *Demonstrationsprototypen* unterstützen die Akquisition von Aufträgen und das Aufsetzen von Projekten.
- *Prototypen im engeren Sinn* sowie *Labormuster* dienen zur Risiko-Minderung und können in jedem Prozessmodell zu jedem Zeitpunkt für diesen Zweck eingesetzt werden.
- Das Arbeiten mit *Pilotsystemen* ist eine Form des Wachstumsmodells.

3.4 Wiederverwendung und Beschaffung

3.4.1 Mehrfachverwendung und Produktivität

Die Produktivität bei der Entwicklung neuer Software lässt sich nach heutigen Erkenntnissen nur in ziemlich eng begrenztem Rahmen steigern, weil wesentliche Teile des Entwicklungsprozesses kreative Arbeiten sind, die nicht automatisierbar sind und nicht durch den Einsatz von Hilfsmitteln massiv beschleunigt werden können (Brooks 1987).

Leistungssteigerungen von mehreren Größenordnungen bei gleichem Preis innerhalb weniger Jahre, wie man dies bei der Hardware beobachten kann, hat es bei der Entwicklung neuer Software nie gegeben und wird es wohl auch nie geben. Der Schlüssel für diesen Unterschied liegt in den verkauften Stückzahlen. *Die Masse macht es*.

Die Kosten für die Neuentwicklung komplexer Hardware, z.B. eines Prozessors, stehen den Kosten für die Entwicklung komplexer Software um nichts nach. Im Gegensatz zur Software aber werden Hardware-Komponenten nach ihrer Entwicklung in großen Stückzahlen verkauft, so dass die Entwicklungskosten pro verkauftem Exemplar drastisch sinken.

Die einzige Möglichkeit, die Kosten pro verkauftem Exemplar Software massiv zu senken, ist eine ebenso massive Steigerung der Stückzahl, in der diese Software verwendet, bzw. verkauft wird. Besonders wirtschaftlich sind also einerseits große Serien eines Produkts mit identischer Software und andererseits die Wiederverwendung von Software.

3.4.2 Wiederverwendung

Wiederverwendung von Software fällt nicht vom Himmel. Sie stellt sich auch nicht automatisch ein, wenn sie als Unternehmensziel postuliert wird, sondern man muss ihr auf die Sprünge helfen. Dies hat technische und organisatorische Gründe.

Technisch muss wiederverwendbare Software zunächst einmal geschaffen werden. Hierzu muss neu erstellte Software so gestaltet, in Komponenten gegliedert und dokumentiert sein, dass entsprechende Komponenten überhaupt herausgelöst und in einem anderen Kontext sinnvoll wiederverwendet werden können.

Im Bereich des Organisatorischen gibt es im Wesentlichen drei Probleme, welche einer breiten Wiederverwendung von Software im Weg stehen:

- Wiederverwendbare Software ist in ihrer Herstellung aufwendiger als Einzweck-Software, weil sie allgemeingültiger angelegt und ausführlicher dokumentiert werden muss. Für jemanden, der nur für ein Projekt verantwortlich ist, besteht daher kein Anreiz, die für das Projekt notwendige Software wiederverwendbar zu gestalten. Wiederverwendbare Software muss deshalb explizit honoriert werden.
- Wiederverwendung scheitert häufig daran, dass Projektleiter oder Projektmitarbeiter nicht wissen, welche im Unternehmen vorhandene Software in ihrem Projekt wiederverwendet werden könnte. Es muss daher Kataloge wiederverwendbarer Software geben.
- Wiederverwendung scheitert häufig daran, dass die Urheber von potenziell Wiederverwendbarem nicht bereit sind, irgendwelche Garantien oder Pflegeverpflichtungen für ihre Software zu übernehmen. Wiederverwendung muss daher wie Beschaffung vertraglich geregelt werden.

Optimal für die Förderung von Wiederverwendung wäre es, wenn es Unternehmen oder unternehmensinterne Profit Center gäbe, die vom Handel mit wiederverwendbaren Software-Komponenten leben. Ansätze in dieser Richtung sind beispielsweise kommerziell vertriebene Klassenbibliotheken und Software-Rahmen (Frameworks).

Eine andere Möglichkeit zur Förderung von Wiederverwendung innerhalb eines Unternehmens ist die Schaffung einer Informationsstelle für Wiederverwendung. Diese trägt Informationen über wiederverwendbare Software zusammen und führt einen Katalog. Interessierte Projektleiter bzw. Entwickler können sich bei der Informationsstelle nach vorhandenen Komponenten für ihr Problem erkundigen.

Wichtig ist, dass beim Sammeln das Holprinzip herrscht: die Informationsstelle sammelt selbst aktiv bei den Entwicklungsabteilungen; diese werden nicht mit zusätzlichem Aufwand belastet (sonst kooperieren sie nicht).

3.4.3 Beschaffung

Wo immer möglich, werden Systeme bzw. Software-Komponenten nicht entwickelt, sondern beschafft. Dies hat rein ökonomische Gründe: aufgrund der verkauften Stückzahlen ist beschaffte Software fast immer kostengünstiger als selbst entwickelte. Allerdings fallen neben den reinen Beschaffungskosten fast immer auch Parametrierungs- und Anpassungskosten an, so dass die Wirtschaftlichkeit einer Beschaffung in jedem Einzelfall geprüft werden muss.

Der Beschaffungsprozess ist typisch in den Software-Entwicklungsprozess eingebettet, weil Spezifikations-, Test- und Installationsarbeiten auch bei Beschaffungen notwendig sind. Die Hauptarbeit ist im Rahmen der Konzipierung der Software zu leisten (siehe Kapitel über Konzipierung).

3.4.4 Wiederverwendung und Beschaffung im Entwicklungsprojekt

Alle Anstrengungen zur Mehrfachverwendung von Software (Wiederverwendung, Verwendung beschaffter Software) sind nutzlos, wenn die Entwicklerinnen und Entwickler nicht mitmachen.

Projektleitende und Entwickelnde dürfen ihren Berufsstolz nicht darin sehen, möglichst alles selbst, neu und perfekt zu machen. Sie müssen vielmehr lernen, dass nicht das Perfekte (und Teure) die beste Lösung ist, sondern das Gebrauchstaugliche, das so weit wie möglich aus vorhandenen Komponenten gefertigt ist.

Die Suche nach Beschaffbarem bzw. Wiederverwendbarem muss zur fest verankerten Tätigkeit in jedem Projekt werden. Damit zeitgerecht entschieden werden kann, sind bereits während der Anforderungsspezifikation entsprechende Vorabklärungen zu treffen.

Die Mitarbeiterinnen und Mitarbeiter dürfen nicht danach beurteilt werden, ob sie möglichst viel Neues produzieren. Sie müssen vielmehr danach beurteilt werden, ob sie Nützliches und Bedarfsgerechtes unter Verwendung von möglichst viel Vorhandenem produzieren.

3.5 Pflege (Wartung)

Pflege, oft auch Wartung (maintenance) genannt, ist der Prozess der systematischen Erhaltung der Gebrauchstauglichkeit von Programmen. Software vom S-Typ (vgl. Abschnitt 3.1.4) braucht keine Pflege, sofern sie richtig entwickelt worden ist. Die nachfolgenden Ausführungen beziehen sich daher auf Software vom P- und E-Typ. Die Pflegemaßnahmen werden in der Regel in drei Klassen unterteilt: Fehlerbehebung, Anpassung und Weiterentwicklung (Erweiterungen, Verbesserungen). Bild 1.3 zeigt die typische Verteilung des Aufwands auf die drei Klassen.

DEFINITION 3.9. *Pflege (Wartung, maintenance).* Modifikation bestehender Software und/oder Ergänzung bestehender Software durch neue Software mit dem Ziel, Fehler zu beheben, die bestehende Software an veränderte Bedürfnisse oder Umweltbedingungen anzupassen oder die bestehende Software um neue Fähigkeiten zu erweitern.

Bei großer Software, die zudem bei mehreren Kunden in Betrieb ist, ist es unmöglich, jede Änderung sofort in Betrieb zu nehmen. Der Verwaltungs- und Installationsaufwand wäre viel zu hoch. Stattdessen werden die Änderungen gebündelt und von Zeit zu Zeit en bloc als neuer *Release* freigegeben (siehe auch Kapitel über Konfigurationsmanagement).

DEFINITION 3.10. *Release.* Eine konsistente Menge von Komponenten eines Systems, die gemeinsam zur Benutzung freigegeben werden.

Bei Software bestehen die Komponenten aus einer Menge von Programm-, Daten- und Steuerkomponenten, die zusammen ein betriebsfähiges System bilden sowie aus Komponenten, welche das System und seine Benutzung dokumentieren.

Meist wird zwischen kleinen und großen Releases unterschieden und dies durch die Nummerierung des Release nach dem Schema m.n kenntlich gemacht. m.n bezeichnet den n-ten kleinen Release innerhalb des m-ten großen Release. Der Abstand zwischen zwei kleinen Releases beträgt in der Regel einige Monate; der zwischen großen Releases ein bis drei Jahre.

Pflegemaßnahmen, auch wenn sie sorgfältig durchgeführt werden, führen typisch dazu, dass die gepflegte Software größer und in ihrem strukturellen Aufbau schlechter wird. Lehman und Belady haben Zahlenmaterial über die Pflege von Software erhoben und daraus sechs Gesetze

abgeleitet (Belady und Lehman 1976, Lehman 1980, Lehman 1992). Alle Gesetze gelten nur für große, während ihrer Lebenszeit ständig benutzte Programme vom P- und vor allem vom E-Typ.

REGEL 3.1. Gesetz der kontinuierlichen Veränderung. Ein Programm unterliegt während seiner gesamten Lebenszeit ständiger Veränderung oder es verliert nach und nach seinen Nutzen. Dieser Prozess schreitet fort, bis es kostengünstiger wird, das Programm durch ein neuerstelltes Nachfolgerprogramm zu ersetzen.

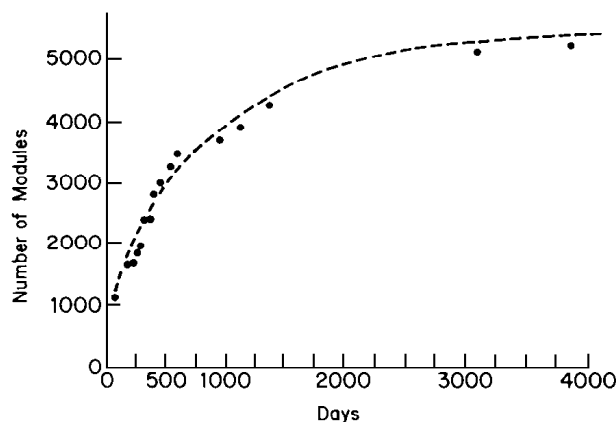
REGEL 3.2. Gesetz der zunehmenden Komplexität. Die Komplexität eines Programms, das ständig verändert wird, nimmt kontinuierlich zu, weil seine Struktur durch die Änderungen immer schlechter wird; es sei denn man ergreift explizite Gegenmaßnahmen.

REGEL 3.3. Grundgesetz der Programm-Evolution. Die Evolution eines Programms unterliegt einer sich selbst regelnden Dynamik mit statistisch bestimmbaren Tendenzen und Invarianzen.

REGEL 3.4. Gesetz der Invarianz des Arbeitsaufwands. Global betrachtet ist der Aufwand, der in die Pflege eines Programms gesteckt wird, während der ganzen Lebenszeit des Programms statistisch konstant.

REGEL 3.5. Gesetz der Invarianz des Release-Inhalts. Der Inhalt der Releases (Änderungen, Ergänzungen, Weglassungen) ist über die Folge aller Releases während der Lebenszeit eines Programms statistisch konstant.

REGEL 3.6. Gesetz des kontinuierlichen Wachstums. Ein Programm muss während seiner gesamten Lebenszeit kontinuierlich wachsen, wenn es gebrauchstauglich bleiben soll.



(Quelle: Lehman und Belady (1985), p. 307)

BILD 3.7. Wachstum des Betriebssystems OS 360/370 von IBM

Während die Mechanismen, die zu den Regeln 3.1, 3.2 und 3.6 führen, intuitiv klar sind, sind die Regeln 3.3 bis 3.5 eher kontra-intuitiv. Intuitiv würde man erwarten, dass der Pflegeprozess durch einschlägige Management-Entscheidungen nachhaltig beeinflussbar ist. Die von Lehman und Belady für mehrere großen Systeme erhobenen Zahlen belegen jedoch das Gegenteil. Das Wachstum folgt klar einer mathematischen Kurve (Bild 3.7), die kumulierte Anzahl geänderter Module folgt einer Geraden, d.h. die Änderungsrate ist konstant (Bild 3.8 a) und die Anzahl der pro Tag geänderten Module, aufgezeichnet über der Folge der Releases, ist statistisch konstant (Bild 3.8 b). Offensichtlich sind die Trägheitsdynamik großer Systeme und Organisationen sowie die begrenzten Fähigkeiten sowohl der Systementwickler wie der Systembenutzer zum Be- und Verarbeiten («Verdauen») von Änderungen so stark, dass sie die Eingriffe von außen marginalisieren, den Pflegeprozess statistisch stabilisieren und ihn selbstregulierend machen.

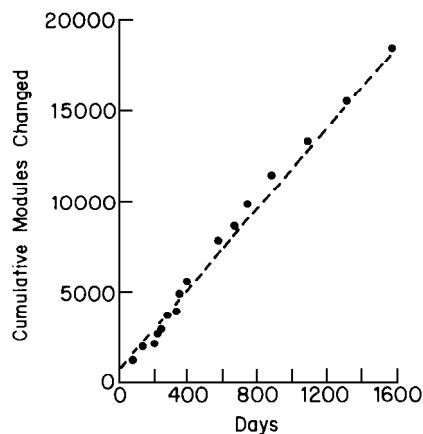
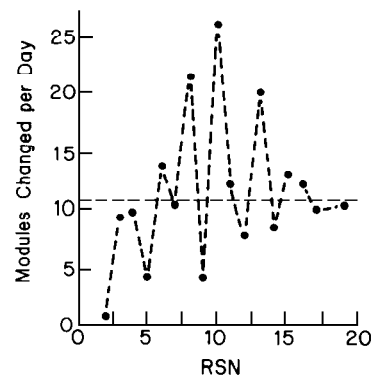


BILD 3.8. A. Kumulierte Anzahl geänderter Module



Quelle: Lehman und Belady (1985), S. 416)

B. Änderungsrate über der Folge der Releases (RSN: Release sequence number)

Aufgaben

- 3.1** Ein zu entwickelndes System besteht aus einer Basiskomponente mit Datenbank und Benutzerschnittstelle sowie sechs voneinander weitgehend unabhängigen Anwendungsfunktionen. Für die Basiskomponente wird ein Entwicklungsaufwand von 12 Personenwochen geschätzt; der Aufwand pro Anwendungsfunktion auf je drei Personenwochen. Sie sind Projektleiterin. Ihr Management verlangt einen Meilenstein mit dem Kriterium «System zu 80% fertig».
- Wie reagieren Sie?
 - Formulieren Sie ein messbares Kriterium für einen solchen Meilenstein.
- 3.2** Gegeben sei das Problem aus Aufgabe 2.5 (Statistischen Auswertung von Betriebsdaten).
- Welches Prozessmodell schlagen Sie für die Entwicklung der geforderten Software vor? Begründen Sie Ihre Entscheidung.
 - Angenommen, Ihre Abteilungsleiterin entscheidet unabhängig von Ihrem Vorschlag, dass dieses Projekt nach einem Wachstumsmodell abzuwickeln ist. Skizzieren Sie einen Lieferplan, indem Sie Anzahl und Reihenfolge der Lieferungen festlegen und für jede Lieferung entscheiden, welche Komponenten der geforderten Lösung in welchem Ausbaustand darin enthalten sein sollen.
- 3.3**
- Wo und mit welchem Ziel kann Prototyping im Software-Entwicklungsprozess eingesetzt werden?
 - Schildern Sie je eine typische Projektsituation, in der die Verwendung eines Demonstrationsprototyps, eines Prototyps im engeren Sinn, eines Labormusters und eines Pilot-systems sinnvoll sind.
- 3.4** Ein Unternehmen betreibt ein selbst entwickeltes System zur Auftragsabwicklung. Die Anwender beklagen sich seit längerer Zeit über die mühsame und nicht mehr zeitgemäße Bedienung des Systems. Mit der Funktionalität des Systems sind die Benutzer nach wie vor zufrieden. Soeben hat die Geschäftsleitung beschlossen, die Benutzeroberfläche des Auftragsabwicklungssystems zu erneuern und hat Sie mit der Durchführung des entsprechenden Projekts beauftragt. Zur Minimierung des Projektrisikos wollen Sie Prototyping einsetzen. Welche Arten von Prototypen setzen Sie wozu ein?
- 3.5** Begründen Sie das Gesetz des kontinuierlichen Wachstums von Lehman (Regel 3.6). Ziehen Sie dabei auch Bild 3.7 heran.

Ergänzende und vertiefende Literatur

Basili und Turner (1975), Boehm (1981, 1988) und Royce (1970) sind die Schlüsselquellen für die vorgestellten Prozessmodelle.

Barnes und Bollinger (1991) diskutieren wirtschaftliche Aspekte der Wiederverwendung und Maßnahmen zur Förderung von Wiederverwendung.

Floyd (1984) gibt eine Taxonomie der verschiedenen Arten des Prototypings; Kieback et al. (1992) geben eine kurze Übersicht über Prototyping als solches und beschreiben dann verschiedene industrielle Prototyping-Projekte.

Lehman (1980) ist der Klassiker über die Evolution von Software.

Das Problem der Definition und der Verbesserung der realen Software-Prozesse in Unternehmen wird in diesem Kapitel nicht behandelt. Interessierte Leserinnen und Leser seien auf Humphrey (1989) verwiesen.

Zitierte Literatur

Barnes, B.H., T.B. Bollinger (1991). Making Reuse Cost-Effective. *IEEE Software* **8**, 1 (Jan. 1991), 13-24.

Basili, V.R., A.J. Turner (1975). Iterative Enhancement: A Practical Technique for Software Development. *IEEE Transactions on Software Engineering* **SE-1**, 6, 390-396.

Boehm, B. (1981). *Software Engineering Economics*. Englewood Cliffs, N.J.: Prentice-Hall.

Boehm, B.W. (1988). A Spiral Model of Software Development and Enhancement. *IEEE Computer* **21**, 5 (May 1988), 61-72.

Brooks, F.P. (1987). No Silver Bullet. Essence and Accidents of Software Engineering. *IEEE Computer* **20**, 4, 10-19.

Floyd, C. (1984). A Systematic Look at Prototyping. In Budde et al. (Hrsg.): *Approaches to Prototyping*. Berlin, etc.: Springer. 1-19.

Gilb, T. (1988). *Principles of Software Engineering Management*. Reading, Mass.: Addison Wesley.

Humphrey, W. S. (1989). *Managing the Software Process*. Reading, Mass.: Addison-Wesley.

IEEE (1990). *Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990. IEEE Computer Society Press.

Kieback A., H. Lichter, M. Schneider-Hufschmidt, H. Züllighoven (1992). Prototyping in industriellen Software-Projekten. *Informatik-Spektrum* **15**, 2 (April 1992). 65-77.

Lehman, M.M. (1980). Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE* **68**, 9, 1060-1076. (Nachgedruckt als Kapitel 19 in Lehman und Belady 1985).

Lehman, M.M., L.A. Belady (Hrsg.) (1985). *Program Evolution: Processes of Software Change*. London, etc.: Academic Press.

Lehman, M.M. (1992). A Model of Software Evolution and its Implications. ABB Computer Science Conference, Kopenhagen.

Mills, H.D., D.O'Neill, R.C. Linger, M. Dyer, R.E. Quinnan (1980). The Management of Software Engineering, Part I-V. *IBM Systems Journal* **19**, 4, 414-477.

Royce, W. (1970). Managing the Development of Large Software Systems. *Proceedings IEEE WESCON*. 1-9. (Nachgedruckt in Proceedings 9th International Conference on Software Engineering, Monterey, 1987, 328-338.)

4. Software-Projektführung

Da Software ein immaterielles Produkt ist, führt eine ad-hoc-Abwicklung bei Software-Projekten wesentlich schneller ins Desaster als bei konventionellen Projekten. Es ist daher notwendig, Software-Projekte systematisch zu führen. Dazu gehören eine sorgfältige Planung, fortlaufende Kontrolle und nötigenfalls Korrektur des Projektablauf. Ferner muss den Projektrisiken besondere Beachtung geschenkt werden.

4.1 Projektplanung

Die Projektplanung schafft die notwendigen Voraussetzungen für das Gelingen eines Projekts. Neben der Definition der zu erreichenden Ziele geht es vor allem darum, den Weg zum Ziel und die benötigten Mittel festzulegen. Dabei sind im Wesentlichen folgende Aufgaben zu erledigen: die Festlegung des zu verwendenden Prozessmodells und der Organisationsstruktur des Projekts, die Bestimmung der Anzahl der benötigten Personen und die Planung ihres Einsatzes, das Aufstellen eines Terminplan und eines Kostenplans, die Festlegung der zu erstellenden Dokumente und der anzuwendenden Verfahren für Entwicklung, Konfigurationsmanagement und Qualitätsmanagement sowie die Abschätzung und Behandlung der Projektrisiken.

4.1.1 Prozessmodell und Organisationsstruktur

Zu den ersten Planungsschritten gehören die Auswahl des Prozessmodells und der Organisationsstruktur für das Projekt auf der Grundlage einer Analyse des Projektkontextes und -inhalts.

Die prinzipielle Struktur der möglichen Prozessmodelle und ihre Eignung in verschiedenen Situationen wurden in Kapitel 3 diskutiert. In der Planung eines konkreten Projekts wird das Prozessmodell konkretisiert und mit einer personellen Organisationsstruktur verbunden. Die personelle Organisationsstruktur legt die Funktionen, Aufgaben, Verantwortlichkeiten und gegenseitigen Beziehungen der am Projekt beteiligten Personen fest.

Die Organisation von Software-Projekten wird hier nicht näher behandelt, da grundsätzlich die gleichen Regeln gelten wie für andere Projekte. Es werden nur zwei Aspekte kurz diskutiert: die Rolle des Projektleiters und die Beziehungen der Projektbeteiligten untereinander.

Die Projektleiterin ist eine Schlüsselfigur in jedem Software-Projekt. Ihre Kompetenzen und Verantwortlichkeiten müssen klar geregelt werden. Anzustreben ist eine *Führung durch Zielsetzung*, d.h. die Auftraggeber bzw. die Verantwortlichen in der Linienorganisation erteilen der Projektleiterin einen *Auftrag* und statten sie mit den dafür notwendigen personellen und materiellen *Ressourcen* aus. Im Rahmen dieses Auftrags handelt die Projektleiterin *eigenverantwortlich*. Sie berichtet der vorgesetzten Stelle regelmäßig über den Projektablauf und informiert insbesondere über Störungen im Ablauf, die mit projekteigenen Mitteln nicht behoben werden können.

Die Beziehungen der Projektbeteiligten untereinander hängen von der Zahl der Personen ab. Bei wenigen Personen ist eine demokratische Struktur möglich, welche Weinberg (1971) idealtypisch als "egoless team" bezeichnet (Bild 4.1 a). Alle Beteiligten sind gleichberechtigt. Probleme und anstehende Entscheidungen werden ausdiskutiert. Die Verantwortung wird gemeinsam getragen. Der Projektleiter ist in dieser Struktur Vorsitzender, aber nicht Chef.

Bei mehr als ca. fünf Beteiligten funktionieren demokratische Teams nicht mehr, weil der Kommunikationsbedarf zu stark steigt (vgl. Bild 1.6). Durch hierarchische Strukturen (ein- oder

zweistufig je nach Personenzahl) wird der Kommunikationsbedarf kanalisiert. Die Projektleiterin steht an der Spitze dieser Hierarchie, ist den Projektbeteiligten gegenüber weisungsbefugt und trägt die Projektverantwortung allein (Bild 4.1 b).

Bei allen Organisationsformen ist anzustreben, dass möglichst viele der Projektbeteiligten möglichst ausschließlich für dieses Projekt arbeiten. Je mehr Projekte eine Person gleichzeitig bearbeitet, desto höher werden die Umschaltverluste beim Übergang von der Arbeit am Projekt X zu der am Projekt Y.

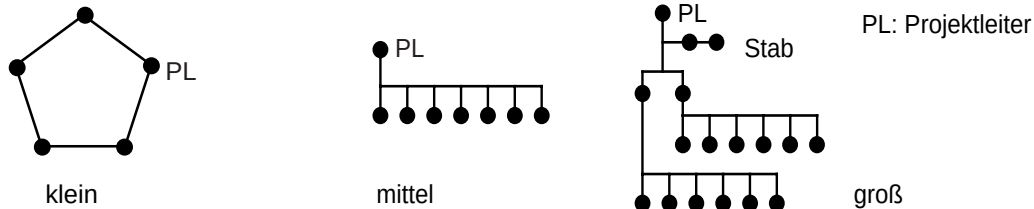


BILD 4.1 A. Demokratisches Team

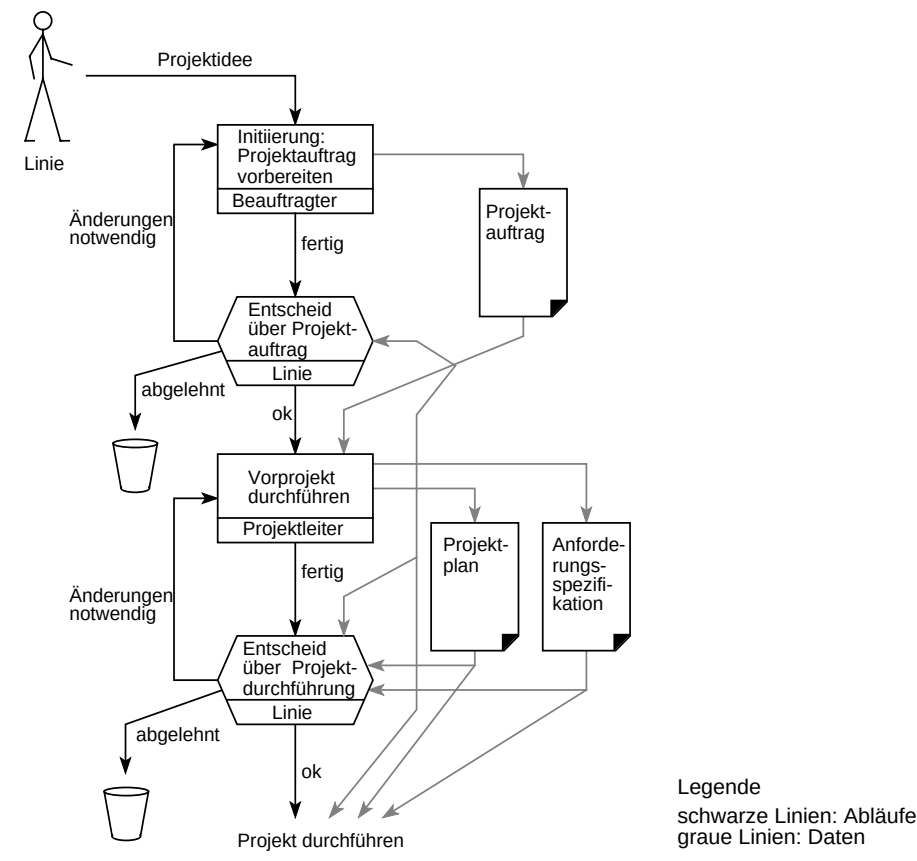
B. Hierarchisch organisiertes Team

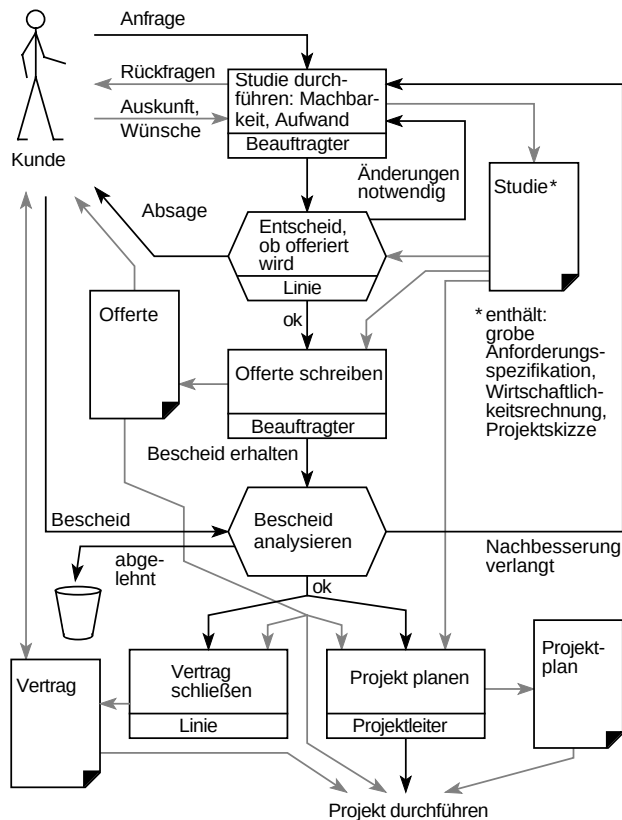
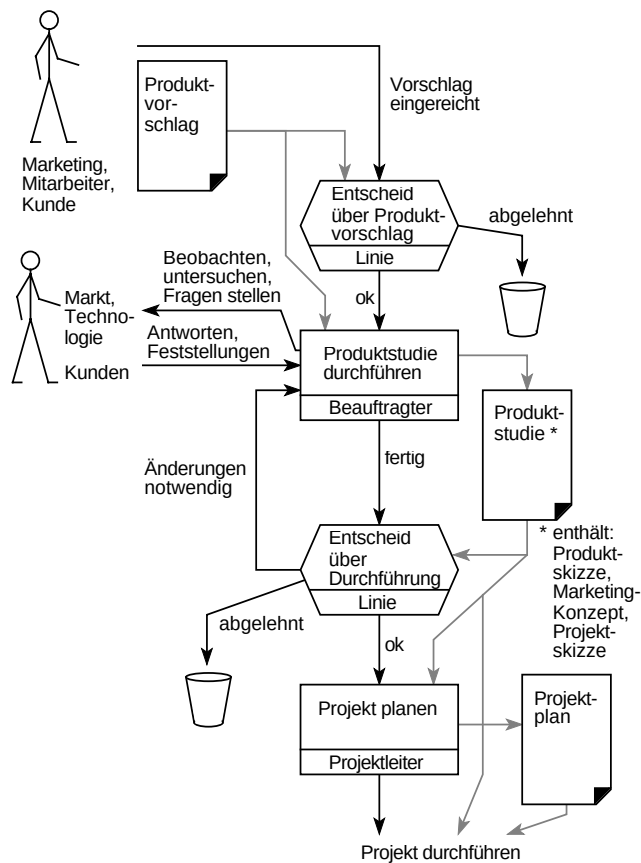
4.1.2 Der Projektstart

Der Projektstart ist eine der kritischen Phasen eines Projekts. Hier werden die Weichen gestellt, die das Projekt auf einen guten Weg oder auf den Holzweg führen.

Es empfiehlt sich, jedem Projekt ein *Vorprojekt* vorzuschalten, in dem das Projekt aufgesetzt wird und wichtige Definitions- und Planungsarbeiten ablaufen. Die Muster 4.1 bis 4.4 zeigen mögliche Modelle für ein Vorprojekt für vier typische Entwicklungssituationen. An Hand dieser Muster wird gezeigt, wo welche Planungsaufgaben anfallen.

MUSTER 4.1. Start eines unternehmensinternen Softwareprojekts



MUSTER 4.2. Start eines externen Auftragsprojekts für Software**MUSTER 4.3. Start eines Produktentwicklungsprojekts für Software**

3.3 Risiken

Liste der Risiken und der geplanten Maßnahmen zu ihrer Steuerung

4. Terminplan

Was wann fertiggestellt sein soll

5. Personalplan**5.1 Projekt-Organigramm**

wie das Projekt personell strukturiert ist

5.2 Personal-Einsatzplan

welche Person welche Aufgabe aus dem Arbeitsplan bearbeitet

6. Kostenplan

geplante Kosten und ihre Verteilung über die Projektdauer

7. Übrige Ressourcen

sonstige benötigte Hilfsmittel (Entwicklungsmaschinen, Software, etc.)

8. Vorgehen

Prozessmodell, Einsatz von Methoden und Werkzeugen, Qualitäts- und Konfigurationsmanagement, zu erstellende Dokumente...

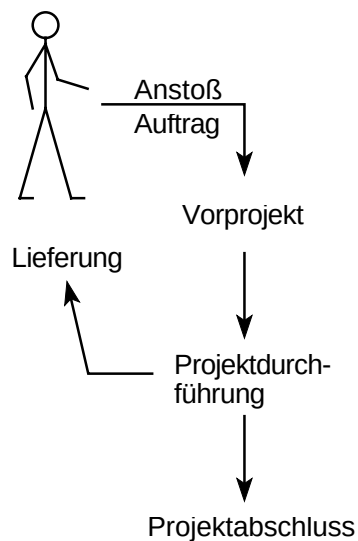
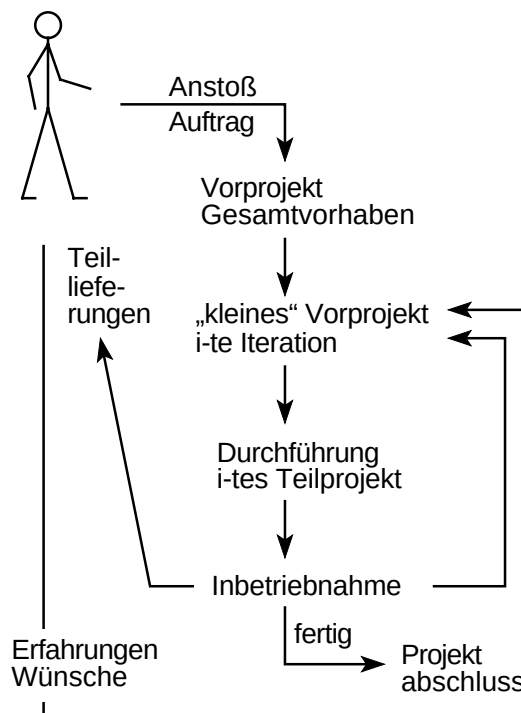
(existiert eine Entwicklungsrichtlinie und/oder ein Q-Plan, so kann das meiste durch Verweis auf diese Dokumente erledigt werden)

4.1.4 Planung bei linearen und bei inkrementellen Prozessen

Bei linearen Prozessmodellen, zum Beispiel dem ergebnisorientierten Phasenmodell, erfolgt die Planung im Vorprojekt. Danach wird die Planung nur noch fortgeschrieben (vgl. 4.2).

Bei Wachstumsmodellen gibt es mehrere Planungsabschnitte. Hier muss zunächst in einem Vorprojekt eine Grobplanung erfolgen, welche insbesondere die Erstellung des Lieferplans umfasst. Jedes Inkrement wird in einem „kleinen“ Vorprojekt dann geplant, wenn es zur Ausführung ansteht. Die Gesamtplanung wird revidiert, wenn es aufgrund der realisierten Inkremente zu Änderungen im Projekt kommt oder wenn unkorrigierbare Abweichungen von der Planung auftreten.

Die Muster 4.6 und 4.7 zeigen die beiden Planungsstile.

MUSTER 4.6. Lineare Planung**MUSTER 4.7.** Inkrementelle Planung

4.1.5 Planungs-Hilfsmittel

Arbeits-, Termin-, Kosten- und Personaleinsatzpläne sollten graphisch in Diagrammen dargestellt werden. Sie werden so angelegt, dass im Projektablauf ein SOLL/IST-Vergleich geführt werden kann.

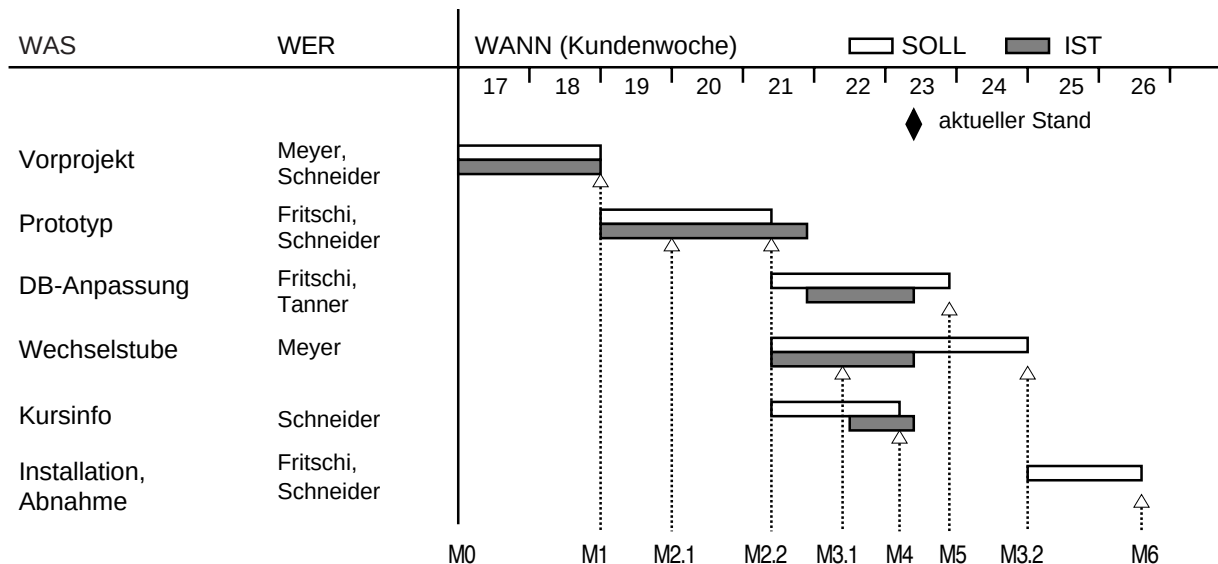


BILD 4.2. Beispiel eines kombinierten Arbeits-/Personaleinsatz-/Terminplans für ein kleines Projekt

Es gibt heute eine Vielzahl von Projektführungs-Werkzeugen für Rechner zu kaufen, mit denen man solche Diagramme erstellen und nachführen sowie SOLL/IST-Vergleiche durchführen kann.

Es ist außerordentlich wichtig, dass die geplanten und die tatsächlichen Aufwendungen möglichst genau erfasst und dokumentiert werden. Denn nur auf der Grundlage verlässlicher Zahlen über abgewinkelte Projekte sind gute Aufwandsabschätzungen für neue Projekte möglich.

4.2 Projektkontrolle und -lenkung

Planung ist nur sinnvoll, wenn auch kontrolliert wird, ob der Arbeitsfortschritt der Planung entspricht. Fortschrittskontrollen müssen messbare Ergebnisse haben, wenn sie nützlich sein sollen. Andernfalls gerät man rasch in die Falle des 90%-Syndroms (Bild 4.3). Die wichtigsten Kontroll-Maßnahmen sind Termin-, Sachziel-, Kosten- und Risikoverfolgung.

Terminverfolgung geschieht mit Hilfe von Meilensteinen. In der Planung hat jeder Meilenstein einen Soll-Termin. Jedesmal, wenn ein Meilenstein erreicht ist, kann durch Vergleich zwischen Soll- und Ist-Termin eine *gesicherte quantitative Aussage* über die Terminalsituation gemacht werden. Jedesmal, wenn ein Solltermin für einen Meilenstein verstreicht, ohne dass der Meilenstein erreicht ist, kann durch eine Schätzung des verbleibenden Aufwands bis zur Erreichung des Meilensteins eine *Schätzung* über die Terminalsituation abgegeben werden.

Sachzielverfolgung geschieht ebenfalls mit Hilfe von Meilensteinen, indem jeder erreichte Meilenstein ein fertiggestelltes Zwischenergebnis dokumentiert.

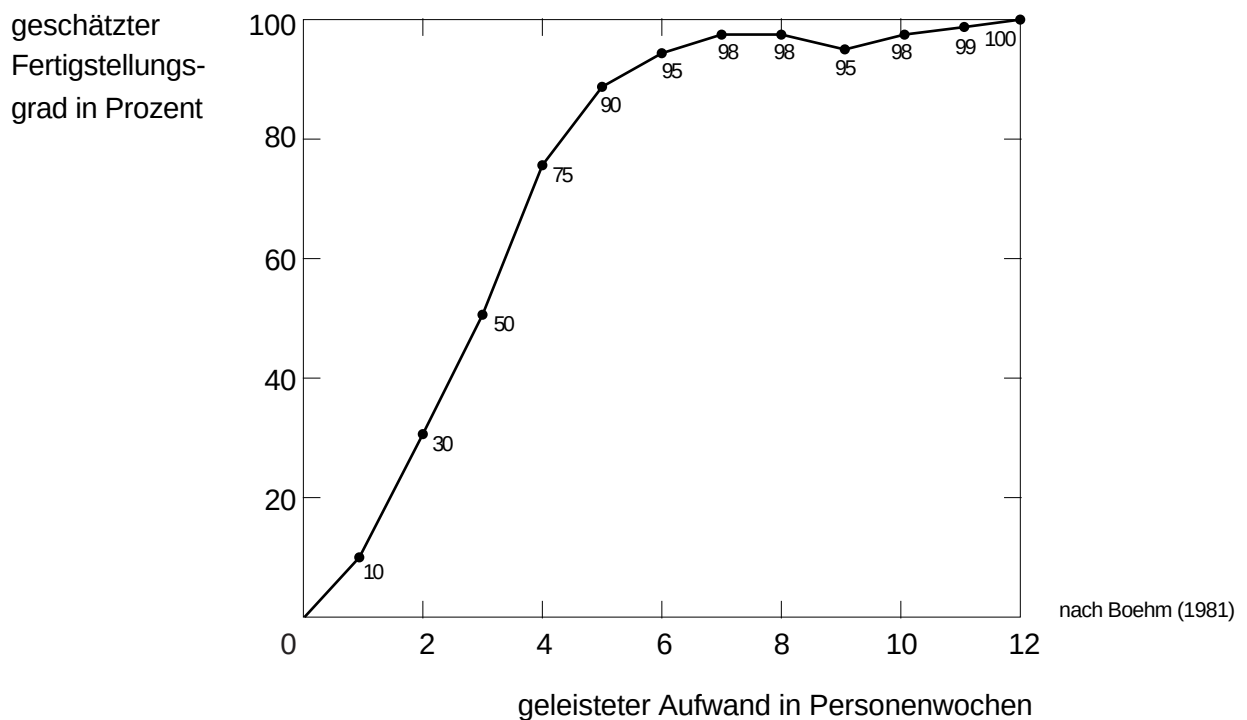


BILD 4.3. Das 90%-Syndrom in der Projektkontrolle

Die scheinbar einfachste Art der *Kostenverfolgung* ist das Aufzeichnen von budgetierten und tatsächlichen Kosten über der Zeit. Bei näherem Hinschauen stellt man aber fest, dass eine solche Graphik für die Projektkontrolle wertlos ist, da der Projektfortschritt nicht berücksichtigt wird. Die gleiche Kurve kann bei einem termingerechten Projekt eine Kostenunterschreitung und bei einem in Verzug geratenen Projekt eine massive Kostenüberschreitung bedeuten. Stattdessen muss entweder der Projektfortschritt als Gewinn bewertet und mit in die Kurve der tatsächlichen Kosten einbezogen werden, oder die Kosten müssen über dem Projektfortschritt statt über der Zeit aufgetragen werden. Ersteres ist aufwendig und wird daher nur selten angewendet. Einfacher ist es, wieder die Meilensteine heranzuziehen und die Kosten über den Meilensteinen aufzutragen. Bei den Ist-Kosten werden dabei die Kosten zum Zeitpunkt der Erreichung des Meilensteins genommen.

Risikoverfolgung ist in Kapitel 4.4 beschrieben.

Ein nützliches Hilfsmittel in der Kontrolle vor allem größerer Software-Projekte ist ein *organisiertes Berichtswesen* (z.B. standardisierte Wochen- und Monats-Fortschrittsberichte). Diese haben im Wesentlichen eine Frühwarnfunktion, indem sie Probleme und Störungen melden, bevor sich diese in der verspäteten Erreichung von Meilensteinen manifestieren.

Bei großen Projekten mit vielen beteiligten Personen haben sich sogenannte *Arbeitspaket-Ordner* (engl. Unit Development Folder) bewährt. In diesen werden alle Ergebnisse sowie alle sonstigen Papiere zu jeweils einem Arbeitspaket eingheftet. Jeder Ordner enthält ein Leitblatt, auf dem die Termine, Verantwortlichkeiten und Arbeitsfortschritte für das zugehörige Arbeitspaket festgehalten sind. Die Arbeitspaket-Ordner können physisch als Ordner oder logisch als Dateien auf einem Rechner geführt werden.

Werden bei der Projektkontrolle größere Abweichungen erkannt, so müssen die Ursachen dafür ergründet und entsprechende *Lenkungsmaßnahmen* ergriffen werden. Bild 4.4 zeigt das Prinzip eines gelenkten Prozesses. Bei Terminüberschreitungen können beispielsweise folgende Lenkungsmaßnahmen in Betracht gezogen werden:

- die Bereitstellung zusätzlicher Ressourcen

- die Befreiung von Projektmitarbeitern von Verpflichtungen außerhalb des Projekts
- die Anordnung von Überstunden
- die Vergabe von Teilaufträgen an Dritte
- Abstriche bei den zu erreichenden Sachzielen
- Etappierung der Sachziele (nach der Art eines Wachstumsmodells).

Wenn erkannt wird, dass der Projektplan trotz Gegenmaßnahmen nicht mehr einhaltbar ist, so muss der Projektplan den neuen Gegebenheiten angepasst werden. Nach unrealistischen Plänen zu arbeiten ist demotivierend und wenig effektiv.

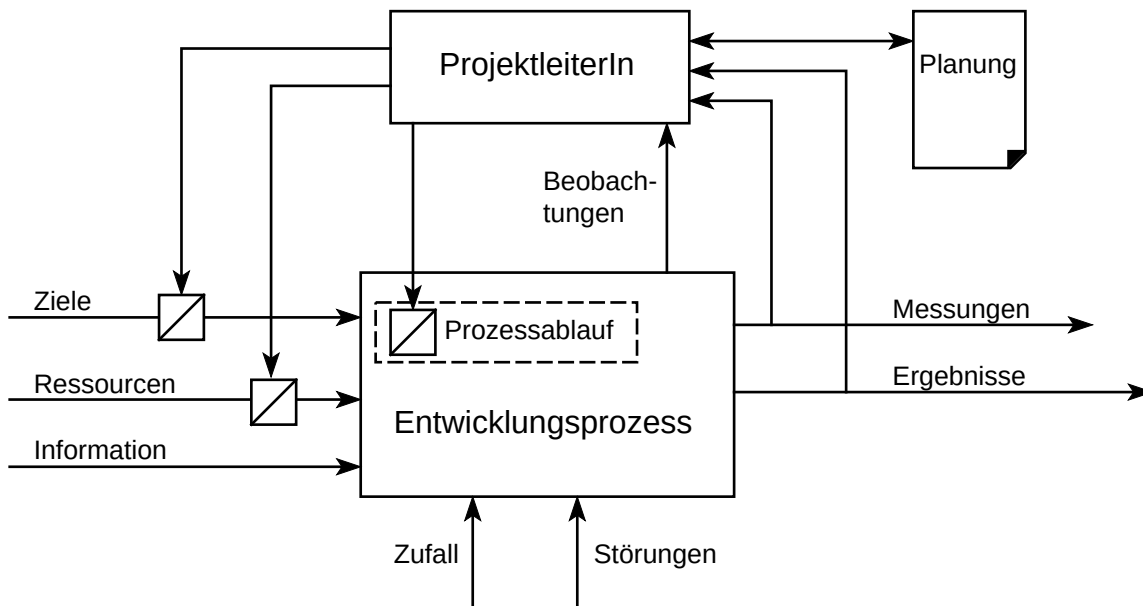


BILD 4.4. Lenkung eines Software-Entwicklungsprojekts

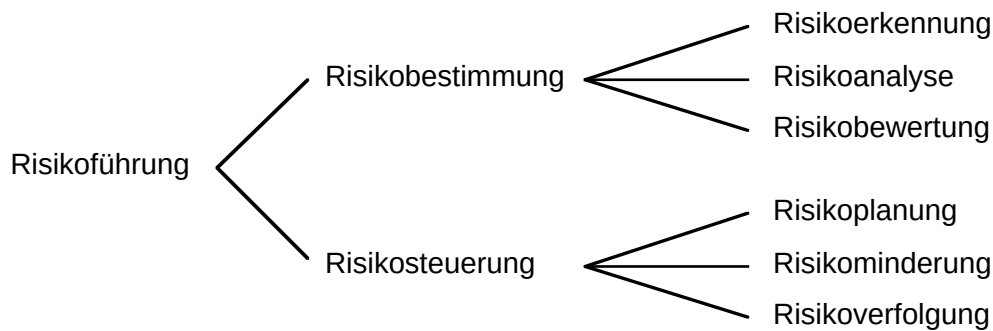
4.3 Der Projektabschluss

Der Projektabschluss ist fast so wichtig wie der Projektstart. Es geht darum, das entstandene Produkt geordnet in die Phase der Pflege überzuleiten und das entstandene Projektwissen zu sichern. Die während des Projekts entstandenen Dokumente und Messungen werden abgeschlossen. Aus den Messwerten werden interessierende Gesamtgrößen bestimmt, zum Beispiel Gesamtaufwand und totale Durchlaufzeit des Projekts. Eine kurze Projektgeschichte hält SOLL und IST bezüglich Sache, Terminen, Kosten und Personalaufwand fest und berichtet über wichtige Erfahrungen, die in zukünftigen Projekten von Nutzen sein könnten.

Alle für die Pflege der Software, für den Nachweis qualitätsrelevanter Eigenschaften oder für eine spätere Auswertung notwendigen Dokumente und Messungen werden archiviert; der Rest wird vernichtet.

4.4 Software-Risikoführung

Projektrisiken sind Ereignisse, welche den sachlichen oder wirtschaftlichen Erfolg eines Projekts bedrohen. Gut geführte Projekte zeichnen sich dadurch aus, dass in ihnen die Risiken nicht dem Zufall überlassen werden, sondern dass die Projektleiterin oder der Projektleiter eine explizite *Risikoführung* betreibt (Boehm 1989, 1991, Charette 1989). Bild 4.5 zeigt die Aufgaben der Risikoführung.

**BILD 4.5.** Aufgaben der Risikoführung

4.4.1 Risikobestimmung

Als erstes müssen die Risiken eines Projekts *erkannt* werden. Eine Checkliste oder zumindest das Durchgehen der Liste der häufigsten Software-Projektrisiken (Tabelle 4.1) ist dabei hilfreich.

TABELLE 4.1. Die 10 häufigsten Software-Risiken (nach Boehm 1991)

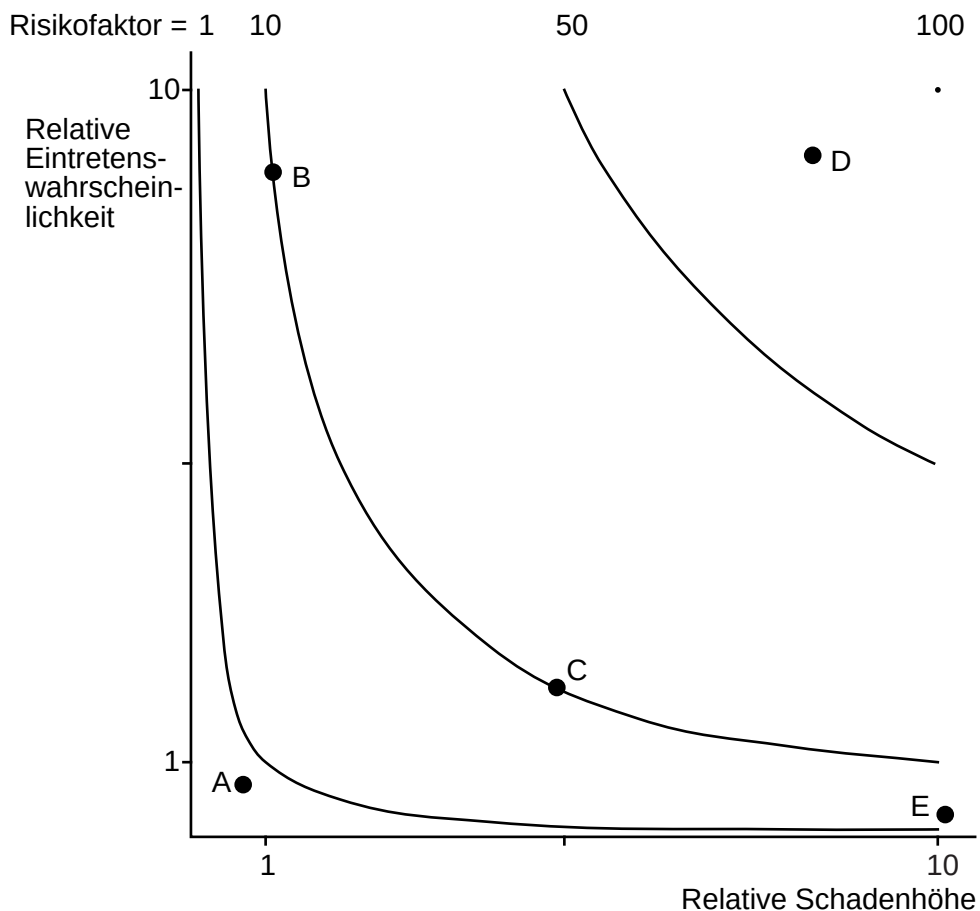
Risiko	mögliche Maßnahmen
zuwenig Leute	gute Leute einstellen, vorhandene Leute ausbilden, Motivation und Arbeitsklima fördern, den richtigen Leuten die richtigen Aufgaben geben
unrealistische Kosten- und Terminpläne	sorgfältige Aufwandschätzung, Entwicklung mit Wachstumsmodell, Anforderungen reduzieren, kostenorientierte Entwicklung
falsche Funktionalität	quantifizierte Ziele, sorgfältige Spezifikation, Prototypen, Beteiligung des Auftraggebers
falsche Benutzerschnittstelle	Prototypen, Einbezug der Endbenutzer (oft nicht identisch mit den Auftraggebern!)
Vergoldung (überflüssiger Luxus)	Kosten-Nutzen-Analyse, Setzen von Prioritäten in den Zielen, kostenorientierte Entwicklung
sich ständig verändernde Anforderungen	Setzen von Wichtigkeits-Schwellwerten (unterhalb derer nicht geändert wird), änderungsfreundlicher Entwurf (z.B. durch Information Hiding), Entwicklung mit Wachstumsmodell
Probleme mit zugekauften Komponenten	sorgfältige Auswahl (z.B. mit Benchmarks), Eingangs-Qualitätskontrolle
Probleme mit extern vergebenen Aufträgen	Überprüfung des Auftragnehmers vor Auftragsvergabe, klar formulierte Aufträge, Zwischeninspektionen während der Abwicklung, Abnahmeinspektion, Aufträge mit Erfolgshonorar
Nichterreichen der verlangten Leistungen (z.B. Reaktionszeit)	Abschätzung in Review, Simulationen, Prototypen, Messung und Optimierung
Überforderung der Mitarbeiter in Bezug auf ihr software-technisches Können	Aufgaben-Analyse, Ausbildung, Reduktion der Anforderungen, Entwicklung mit Wachstumsmodell

In der *Risikoanalyse* wird die Gefährlichkeit der einzelnen Risiken bestimmt und werden Probleme untersucht, die sich aus dem gleichzeitigen Eintreten mehrerer Risiken ergeben.

Die Gefährlichkeit eines Risikos ist abhängig von zwei Größen: der Eintretenswahrscheinlichkeit und der Schadenhöhe. Es ist daher sinnvoll, Risiken nach ihrem *Risikofaktor* zu beurteilen. Dieser ist das Produkt aus Eintretenswahrscheinlichkeit und Schadenhöhe (Bild 4.6). Üblicherweise werden relative Skalen verwendet, bei einfachen Abschätzungen nur mit den Werten gering, mittel und hoch, bei genauerem Arbeiten mit Werten von 0 bis 10.

Das Arbeiten mit Risikofaktoren hat allerdings einen Schönheitsfehler: Risiken mit sehr hohem Schaden bei gleichzeitig sehr geringer Eintretenswahrscheinlichkeit (sogenannte *Restrisiken*) können einen sehr kleinen Risikofaktor haben. Hier kommen andere Überlegungen zum Zug, zum Beispiel ob die Firma einen solchen Schaden finanziell überlebt oder ob der Schaden sozial noch tolerierbar ist.

Bei Risikokombinationen können die resultierenden Risikofaktoren anhand eines Entscheidungsbaums bestimmt werden.



- A ist ein harmloses Risiko
- Die Risiken B und C haben den gleichen Risikofaktor und sind somit etwa gleich gefährlich
- D ist ein sehr gefährliches Risiko, welches unbedingt unter Kontrolle gebracht werden muss
- E ist ein sogenanntes Restrisiko, welches anders beurteilt werden muss (siehe Text)

BILD 4.6. Beurteilung von Risiken

In der *Risikobewertung* werden die Konsequenzen aus der Risikoanalyse gezogen. Es wird eine Rangliste der Risiken erstellt und es werden Maßnahmen zur Risikominderung geplant und eingeleitet.

4.4.2 Risikosteuerung

Um die erkannten Risiken in den Griff zu bekommen, braucht es für jedes größere Risiko einen *Plan*, wie das Risiko beherrscht werden soll. Grundsätzlich ist bei jedem Risiko zu überlegen

- ob das Risiko vermeidbar ist
- welche Maßnahmen zur Risikominderung getroffen werden sollen
- wie das Risiko im Projekt verfolgt werden soll
- ob das Risiko auf Dritte abgewälzt werden kann.

Tabelle 4.1 zeigt eine Reihe von Maßnahmen zur *Risikominderung*. Sie lassen sich mit folgenden Stichworten zusammenfassen: Informationsbeschaffung, Personalpolitik, Prototypen, Simulationen, Vereinfachung.

Bei der *Risikoverfolgung* geht es darum, die wichtigsten Risiken und ihre Entwicklung während der gesamten Projektabwicklung im Auge zu behalten. Boehm zum Beispiel empfiehlt, in jedem Projekt eine Rangliste der 10 wichtigsten Risiken zu führen. In den in regelmäßigen Abständen stattfindenden Sitzungen der Projektverantwortlichen mit ihren Vorgesetzten wird diese Liste besprochen und aktualisiert. Die dynamische Entwicklung der Rangliste bildet die Grundlage, um Maßnahmen zur Risikominderung zu treffen oder getroffene Maßnahmen zu überprüfen und gegebenenfalls anzupassen.

4.4.3 Wirtschaftlichkeit der Risikoführung

Eine sorgfältige Risikoführung kann mit erheblichen Kosten verbunden sein. Fragt man sich nach der Wirtschaftlichkeit dieser Aufwendungen, so ist folgendes zu beachten. Risikoführungskosten amortisieren sich bei jedem Projekt, bei dem die Risikoführung das Eintreten eines größeren Risikos verhindert oder dessen Schaden erheblich mindert. Einen entsprechenden Nachweis zu führen, ist im Einzelfall allerdings oft schwierig. In Projekten, in denen es rückblickend keine Probleme mit Risiken gab, sind die Risikoführungskosten ähnlich zu betrachten wie Versicherungsprämien: Sie sind, in vernünftigem Umfang eingesetzt, im statistischen Mittel wirtschaftlich und sie vermindern drastisch die Wahrscheinlichkeit, dass der Risikofall zur finanziellen Katastrophe wird.

Aufgaben

- 4.1** Gegeben sei das Problem aus Aufgabe 2.5 (System zur statistischen Auswertung von Betriebsdaten). Zum Umfeld sei folgendes zusätzlich bekannt: Sie verfügen als ProjektleiterIn über eine erfahrene Gruppe von Entwicklerinnen und Entwicklern. Allerdings hat die Informatikerin, welche bei der Entwicklung des Vorgängersystems die Konzepte erstellt und die Hauptarbeit geleistet hatte, vor einem Monat das Unternehmen verlassen. Wegen Sparmaßnahmen in Ihrer Firma hat Ihre Chefin die Stelle bisher nicht wiederbesetzen dürfen. Einen Ihrer Projektmitarbeiter müssen Sie für ein strategisches Projekt an eine andere Abteilung ausleihen. Sie verfügen damit über rund 30% weniger Arbeitskapazität, als Sie für die Abwicklung Ihres Projekts eigentlich brauchen. Eine Analyse ergibt folgende Aufwandschätzung für das Projekt:
- Portierung der vorhandenen Software mit gleichzeitiger Trennung in einen maschinenabhängigen Infrastrukturteil und einen maschinenunabhängigen Applikationsteil: 12 Personenwochen
 - Parametrierung des verarbeitbaren Datenvolumens: 3 Personenwochen
 - Erstellung der graphischen Auswertungen auf der Basis einer gekauften Graphik-Bibliothek: 12 Personenwochen
 - Abnahme und Projektabschluss: 1 Personenwoche.

Zur Problemanalyse, Planung und Erstellung eines Rahmenkonzepts für die Lösung haben Sie bisher zwei Wochen aufgewendet; eine weitere Woche werden Sie noch brauchen. Der vereinbarte Liefertermin ist heute in neun Wochen. Sie können (sich selbst eingerechnet) maximal drei Leute mit voller Arbeitskapazität für das Projekt einsetzen.

Entwerfen Sie einen Aufgaben- und Terminplan für dieses Projekt im Stil von Bild 4.2.

- 4.2** Begründen Sie, warum das Auftragen von budgetierten und tatsächlichen Kosten über der Zeit ein unbrauchbares Mittel für die Kostenverfolgung in Projekten ist.
- 4.3** Gegeben sei das Problem aus Aufgabe 4.1; es ist aber 10 Tage früher. Sie haben soeben mit der Problemanalyse begonnen und wollen nun die Risiken des Projekts abschätzen.
- a) Welches sind Ihrer Ansicht nach die größten Risiken, die den Erfolg Ihres Projektes gefährden können?
- b) Welche Maßnahmen können Sie ergreifen, um diese Risiken zu steuern und/oder zu verkleinern?

Ergänzende und vertiefende Literatur

Es gibt eine Fülle von Literatur über Software-Projektmanagement. Stellvertretend seien hier die folgenden zwei Bücher herausgegriffen.

Frühauf, Ludewig und Sandmayr (1999) geben eine kurz gefasste Einführung in die Probleme der Software-Projektführung.

Weinberg (1992) gehört zur Pflichtlektüre für angehende Software-Manager und -Projektleiter.

Glinz (1999) enthält eine Standortbestimmung sowie weitere Literaturverweise.

Boehm (1991) gibt einen Überblick über die Problematik der Risikoführung. Charette (1989) ist ein Lehrbuch über Risikoführung. Boehm (1989) ist ein Tutorium, das Artikel verschiedener Autoren zur Risikoproblematik enthält.

Zitierte Literatur

Boehm, B.W. (1989). *Software Risk Management*. Los Alamitos, Ca.: IEEE CS Press.

Boehm, B.W. (1991). Software Risk Management: Principles and Practice. *IEEE Software* **8**, 1 (Jan. 1991), 32-41.

Charette, R.N. (1989). *Software Engineering Risk Analysis and Management*. New York, etc.: McGraw-Hill.

Frühauf, K., Ludewig, J., Sandmayr, H. (1999). *Software-Projektmanagement und -Qualitätssicherung*. Dritte, überarbeitete Auflage. Zürich: vdf.

Glinz, M. (Hrsg.) (1999). *Software-Projektmanagement*. *Informatik/Informatique* **6**, 5 (Okt 1999).

Weinberg, G.M. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.

Weinberg, G.M. (1992). *Quality Software Management: Volume 1, Systems Thinking*. New York: Dorset House.

5. Software-Aufwandschätzung

5.1 Allgemeines

Bild 5.1 zeigt, wie man es nicht machen sollte: Eine erste Aufwandschätzung wird zunächst nach unten korrigiert, um den Auftrag zu bekommen. Anschließend wird die Schätzung jedesmal dann nach oben angepasst, wenn sie von den tatsächlichen Kosten eingeholt worden ist. Die Schätzung nach sieben Monaten ist offenbar die erste, die wirklich seriös durchgeführt wurde. Sie zeigt, dass schon die erste Schätzung viel zu optimistisch war, ganz zu schweigen von der politischen „Schätzung“ nach drei Monaten.

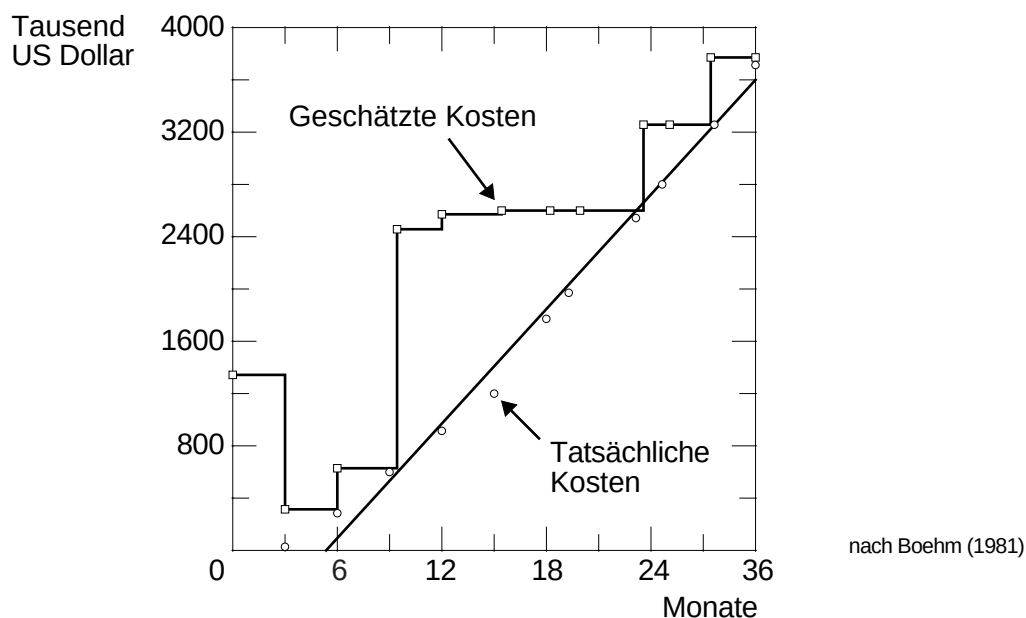
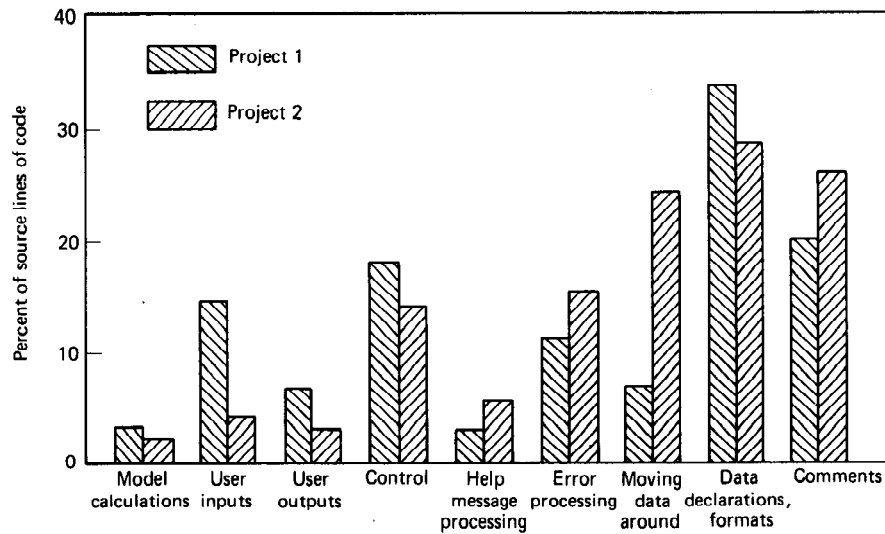


BILD 5.1. Unrühmliches Beispiel für die Kostenentwicklung eines Projekts

Es gibt viele Gründe für die zum Teil krassen Fehleinschätzungen von Software-Kosten bzw. -Terminen, u.a.

- Software-Erstellung ist weitgehend Kopfarbeit und daher stark von der Leistungsfähigkeit dieser Köpfe abhängig. Diese Leistungsfähigkeit kann um mehr als eine Größenordnung schwanken.
- Nur ein kleiner Teil einer Software trägt die eigentliche Funktionalität. Ein großer Teil des Aufwands (und das wird oft übersehen) geht in Verwaltung, Fehlerbehandlung, Benutzerschnittstelle, etc. (Bild 5.2).
- Erfahrungen aus kleinen Projekten werden linear extrapoliert (Bild 1.7).
- Programmierer programmieren nicht 100% ihrer Zeit.



Quelle: Boehm (1981)

BILD 5.2. Woraus ein Programm besteht

5.2 Empirische Schätzverfahren

5.2.1 Expertenbeurteilung

Expertenbeurteilung ist eine vornehme Bezeichnung für Schätzungen über den Daumen. Die Güte der Schätzung steht und fällt mit der Erfahrung der Schätzenden. In der Regel wird aufgrund von Analogien zu bisher abgewickelten Projekten geschätzt. Dies geht bei erfahrenen Leuten in der Regel gut, wenn Erfahrungen mit analogen Projekten vorliegen. Krasse Fehler ergeben sich oft dann, wenn Erfahrungen fehlen oder Erfahrungen mit kleinen Projekten auf große Projekte extrapoliert werden (Bild 1.7).

FESTSTELLUNG 5.1. Expertenbeurteilung ist ein einfaches und billiges Schätzverfahren, das dann recht gut funktioniert, wenn die Schätzenden Erfahrungen mit gleichartigen Projekten haben. Andernfalls können die Prognosen sehr ungenau sein.

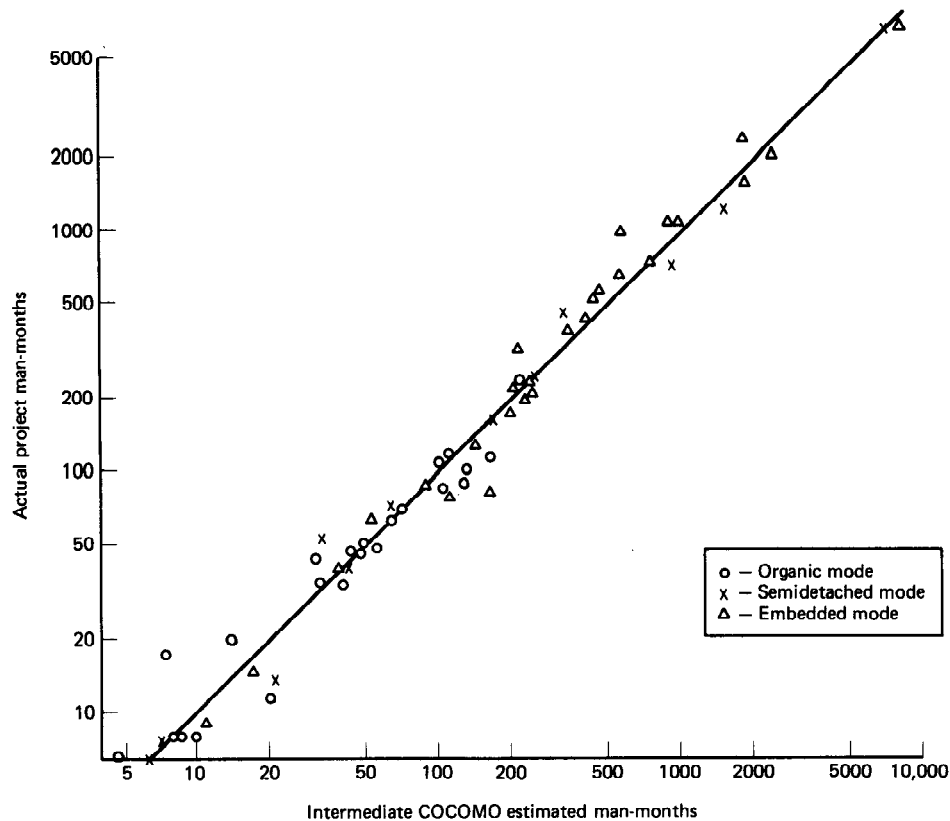
5.2.2 Delphi-Methode

Die Delphi-Methode versucht, Expertenschätzungen zu objektivieren. Mehrere Personen geben unabhängig voneinander eine begründete Schätzung ab. In einer nächsten Runde erhält jede(r) Beteiligte eine Zusammenfassung der ersten Schätzungen. Alle erstellen daraufhin eine neue Schätzung, wobei Abweichungen vom Mittelwert der vorhergehenden Runde zu begründen sind. Dies geht über mehrere Runden, bis alle Schätzungen einigermaßen beieinander liegen.

FESTSTELLUNG 5.2. Die Delphi-Methode liefert zuverlässigere Schätzungen als die Expertenbeurteilung, weil sie Ausreißer eliminiert. Als Nachteil muss ein erheblich höherer Schätzaufwand in Kauf genommen werden.

5.3 Algorithmische Schätzverfahren

Algorithmische Methoden bestehen aus einem oder mehreren Algorithmen zur Berechnung einer Kosten- bzw. Durchlaufzeit-Funktion aus einer Reihe von Variablen. Bei hinreichend genauen Eingaben ergeben sich erstaunlich präzise Prognosen (Bild 5.3).



Quelle: Boehm (1981)

BILD 5.3. Genauigkeit von COCOMO-Schätzungen

Die Genauigkeit aller algorithmischen Methoden hängt jedoch entscheidend von zwei Dingen ab:

- die *Eingangsgrößen* der Kostenfunktion (z.B. Anzahl Instruktionen) müssen einigermaßen *zutreffend geschätzt* werden
- das Modell muss *kalibriert* werden, d.h. der Wert der einzelnen Kostenattribute muss an die jeweilige Entwicklungsumgebung angepasst werden. Eine solche Kalibrierung ist nur möglich, wenn genügend Messwerte von durchgeführten Projekten vorliegen.

FESTSTELLUNG 5.3. Algorithmische Methoden liefern die besten Schätzungen. Sie können aber nur nach entsprechenden Vorarbeiten (Kalibrierung) überhaupt eingesetzt werden. Außerdem sind sie stark abhängig von der Genauigkeit, mit der die Eingangsgrößen bestimmt werden können (garbage-in-garbage-out-Problem).

5.4 Beispiele algorithmischer Schätzverfahren

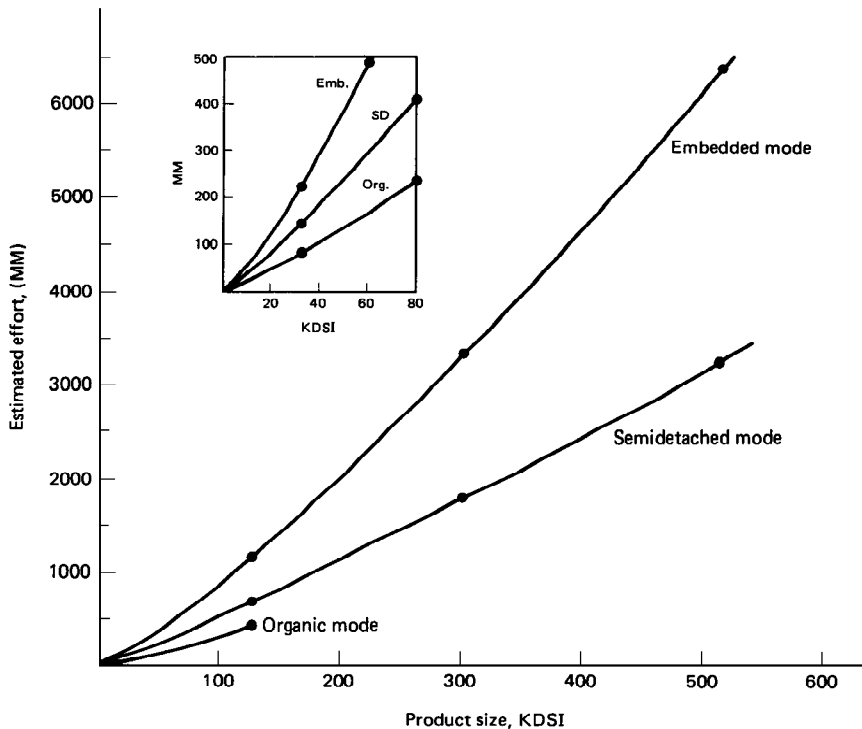
Die zwei wichtigsten algorithmischen Schätzverfahren, COCOMO und Function Point, werden in den folgenden beiden Unterkapiteln vorgestellt.

5.4.1 COCOMO

COCOMO (Constructive Cost Model) ist das Kostenschätzverfahren von Boehm (1981). Es geht aus von einer Schätzung der Produktgröße in KDSI (Kilo lines of delivered source instructions). Aus diesem Grundwert und einer Reihe von Kosten-Multiplikatoren werden Kosten und Durchlaufzeit berechnet. Die Grundgleichungen lauten:

$$(5.1) \quad MM = 2.4 \text{ KDSI}^{1.05} \quad (MM = \text{man month})$$

$$(5.2) \quad TDEV = 2.5 \text{ MM}^{0.38} \quad (TDEV = \text{time to develop})$$



Quelle: Boehm (1981)

BILD 5.4. Aufwandsschätzung nach COCOMO

Diese Gleichungen gelten für einfache *Anwendungsprogramme* (organic mode). Für *Programmsysteme* (semidetached mode), bei denen ein erheblicher Anteil an Interaktion mit Betriebssystem, Gerätetreibern, etc. hinzukommt und für *eingebettete Systeme* (embedded mode), deren Erstellung nochmals erheblich schwieriger ist, gelten andere Faktoren.

COCOMO-Grundgleichungen für Programmsysteme

$$(5.3) \quad MM = 3.0 \text{ KDSI}^{1.12}$$

$$(5.4) \quad TDEV = 2.5 \text{ MM}^{0.35}$$

COCOMO-Grundgleichungen für eingebettete Systeme

$$(5.5) \quad MM = 3.6 \text{ KDSI}^{1.2}$$

$$(5.6) \quad TDEV = 2.5 \text{ MM}^{0.32}$$

Bild 5.4 zeigt eine graphische Veranschaulichung der Kosten über der Produktgröße.

Die mit den Grundgleichungen ermittelten Nominalwerte können nun noch erheblich genauer gemacht werden, indem man sie mit einer Reihe von Kostenfaktoren multipliziert. Bild 5.5 zeigt Boehm's Kostenfaktoren-Tabelle. Die aktuellen Werte aller Kostenfaktoren für ein Projekt werden unabhängig voneinander geschätzt und anschließend alle miteinander multipliziert. Der

Nominalwert für den Projektaufwand in Personenmonaten wird dann mit diesem Produktfaktor multipliziert.

$$(5.7) \quad MM_{\text{Korr}} = \text{Produkt der Kostenfaktoren} \times MM_{\text{nominal}}$$

Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
RELY Required software reliability	.75	.88	1.00	1.15	1.40	
DATA Data base size		.94	1.00	1.08	1.16	
CPLX Product complexity	.70	.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME Execution time constraint			1.00	1.11	1.30	1.66
STOR Main storage constraint			1.00	1.06	1.21	1.56
VIRT Virtual machine volatility ^a		.87	1.00	1.15	1.30	
TURN Computer turnaround time		.87	1.00	1.07	1.15	
Personnel Attributes						
ACAP Analyst capability	1.46	1.19	1.00	.86	.71	
AEXP Applications experience	1.29	1.13	1.00	.91	.82	
PCAP Programmer capability	1.42	1.17	1.00	.86	.70	
VEXP Virtual machine experience ^a	1.21	1.10	1.00	.90		
LEXP Programming language experience	1.14	1.07	1.00	.95		
Project Attributes						
MODP Use of modern programming practices	1.24	1.10	1.00	.91	.82	
TOOL Use of software tools	1.24	1.10	1.00	.91	.83	
SCED Required development schedule	1.23	1.08	1.00	1.04	1.10	

Quelle: Boehm (1981)

BILD 5.5. COCOMO-Kostenfaktoren

Eine ausführliche Beschreibung von COCOMO, inklusive der Definition, was bei ihm ein Personenmonat und eine Zeile gelieferter Code ist, findet sich in Boehm (1981).

5.4.2 Das Function Point-Verfahren

Function Points sind ein *relatives Maß zur Bewertung der Funktionalität*, d.h. des Leistungsumfangs eines Systems. Verfügt ein Unternehmen über Erfahrungszahlen, wieviel Aufwand pro Function Point im Mittel benötigt wird, um Software zu entwickeln bzw. zu pflegen, so können Function Points zur Aufwandschätzung herangezogen werden. Wird in laufenden oder abgeschlossenen Projekten der mittlere Zeitbedarf pro Function Point bestimmt, so bekommt man ein Produktivitätsmaß.

Das Function Point-Verfahren wurde von Albrecht (1979) bei IBM entwickelt und seither von verschiedenen Autoren bzw. Gremien ergänzt und weiterentwickelt (Seibt 1987, Symons 1988, IFPUG 1994). Das Verfahren basiert auf der Idee, die folgenden Größen eines Software-Systems in geeigneter Weise zu zählen und zu bewerten (in Klammern die Terminologie von Albrecht):

- Dateneingaben (External input)
- Datenausgaben (External output)
- Anfragen (External inquiry)
- Schnittstellen zu externen Datenbeständen (External interface file)
- Interne Datenbestände (Logical internal file)

Dateneingaben, Datenausgaben und Anfragen werden an Hand der logischen Transaktionen, die das untersuchte System ausführen soll, gezählt. Tauchen die gleichen Eingabe- bzw. Ausga-

bedaten in verschiedenen Transaktionen mit unterschiedlicher Verarbeitungslogik auf, werden sie mehrmals gezählt. Bei den Datenbeständen werden logische Dateien bzw. logische Daten-
gruppen (Entitätstypen, Relationen) in Datenbanken gezählt. Die Werte können alle bereits in
der Anforderungsspezifikation gezählt werden, sofern diese hinreichend vollständig und detail-
liert ist.

Jede Dateneingabe, Datenausgabe, Anfrage, etc. wird als „einfach“, „mittel“ oder „komplex“
bewertet und mit einem entsprechenden Gewicht versehen. Tabelle 5.1 zeigt als Beispiel die Ge-
wichtungskriterien für Dateneingaben; Tabelle 5.2 zeigt ein Schema für die Berechnung des
Function Point-Rohwerts (unadjusted Function Points, UFP). Hat also beispielsweise ein System
37 logische Transaktionen mit Dateneingaben, von denen 12 als einfach, 16 als mittel und 9 als
komplex bewertet werden, so ergibt sich ein Wert von $12 \times 3 + 16 \times 4 + 9 \times 6 = 154$ als Wert für die
Dateneingaben.

TABELLE 5.1. Gewichtungskriterien für Dateneingaben (IFPUG 1994)

Anzahl bearbeiteter Datenbestände	Anzahl unterscheidbarer Datenelemente in der Eingabe		
	1–4	5–15	>15
0–1	einfach	einfach	mittel
2	einfach	mittel	komplex
>2	mittel	komplex	komplex

TABELLE 5.2. Schema zur Berechnung des Function Point Rohwerts UFP (IFPUG 1994)

Element	Schwierigkeitsgrad			Summe
	einfach	mittel	komplex	
Dateneingaben	_____ x 3 = _____	_____ x 4 = _____	_____ x 6 = _____	_____
Datenausgaben	_____ x 4 = _____	_____ x 5 = _____	_____ x 7 = _____	_____
Anfragen	_____ x 3 = _____	_____ x 4 = _____	_____ x 6 = _____	_____
Ext. Schnittstellen	_____ x 5 = _____	_____ x 7 = _____	_____ x10 = _____	_____
Int. Datenbestände	_____ x 7 = _____	_____ x10 = _____	_____ x15 = _____	_____
Function Point Rohwert (UFP)				_____

Der Rohwert wird mit Gewichtungsfaktoren multipliziert, welche die technische Komplexität
des Systems reflektieren. Der sogenannte Wertkorrekturfaktor (VAF, value adjustment factor¹)
berechnet sich nach der Formel

$$(5.8) \quad VAF = 0,65 + 0,01 \times TDI$$

TDI ist der “total degree of influence”, der gemäß Tabelle 5.3 berechnet wird. Die Formel ist
so konstruiert, dass VAF in einem Bereich zwischen 0,65 und 1,35 liegt. Die Function Points FP
eines Systems berechnen sich dann zu

$$(5.9) \quad FP = UFP \times VAF$$

Sollen Function Points zur Aufwandschätzung verwendet werden, so muss der zu erwartende
Aufwand pro Function Point bekannt sein. Entsprechende Kurven oder Tabellen sind im Umlauf
(Bild 5.6), sind aber mit Vorsicht zu genießen. Der Aufwand pro Function Point ist stark von
projektspezifischen und unternehmensspezifischen Faktoren abhängig, beispielsweise vom Kön-
nen der Leute im betreffenden Unternehmen, der verwendeten Entwicklungsumgebung und dem
Stellenwert von Qualität (z.B. Umfang der verlangten Dokumente, mittlerer Testaufwand). Fer-

¹ In manchen Publikationen heißt dieser Faktor auch TCF (Technical Complexity Factor)

ner muss geklärt werden, welche Aufgaben in der Aufwandschätzung enthalten sind (beispielsweise, ob der Aufwand für die Anforderungsspezifikation eingeschlossen ist oder nicht). Wie Bild 5.6 zeigt, können schon innerhalb desselben Unternehmens signifikant unterschiedliche Umrechnungskurven resultieren.

TABELLE 5.3. Schema zur Bestimmung der Einflussfaktoren (IFPUG 1994)

Nr.	Faktor	Wert	Einzusetzen sind Werte zwischen 0 und 5 0 nicht vorhanden, kein Einfluss 1 unbedeutender Einfluss 2 mäßiger Einfluss 3 durchschnittlicher Einfluss¹⁾ 4 erheblicher Einfluss 5 starker Einfluss ¹⁾Von der Konstruktion des Maßes her müsste eigentlich VAF = 1 gelten, wenn alle Faktoren durchschnittlichen Einfluss haben. Das wäre der Fall, wenn durchschnittlicher Einfluss mit 2.5 statt mit 3 bewertet würde. Aus Praktikabilitätsgründen hat man sich aber für den Wert 3 entschieden, was dazu führt, dass VAF bei lauter durchschnittlichen Einflüssen den Wert 1.07 hat.
1	Datenkommunikation		
2	Verteilte Funktionen		
3	Leistungsanforderungen		
4	Belastung der Hardware		
5	Verlangte Transaktionsrate		
6	Online-Dateneingabe		
7	Effiziente Benutzerschnittstelle		
8	Online-Datenänderungen		
9	Komplexe Verarbeitungen		
10	Wiederverwendbarkeit		
11	Einfache Installation		
12	Einfache Benutzbarkeit		
13	Installation an mehreren Orten		
14	Änder- und Erweiterbarkeit		
Summe der Faktoren (TDI)			

Jones (1996) gibt Faustregeln zur Berechnung des Aufwands aus Function Points an.

REGEL 5.1. *Faustregeln von Jones zur Aufwandberechnung.*

- A. Durchlaufzeit [in Monaten] = $FP^{0.4}$
- B. Anzahl Mitarbeiter = $FP / 150$
- C. Aufwand = Durchlaufzeit x Anzahl Mitarbeiter = $FP^{0.4} \times FP / 150$

Sollen projektspezifische Faktoren (zum Beispiel die Fähigkeit der beteiligten Personen, ihre Vertrautheit mit Problemstellung und die Vertrautheit mit der Lösungsplattform) berücksichtigt werden, so müssen bei der Umrechnung von Function Points in Personenmonate zusätzliche Korrekturfaktoren zur Anwendung kommen.

Das Function Point-Verfahren muss folglich wie jedes andere algorithmische Verfahren kalibriert werden, wenn damit brauchbare Prognosen erzielt werden sollen. Das bedeutet, dass abgeschlossene Projekte nachkalkuliert werden müssen. Aus diesen Daten müssen unternehmens- und eventuell auch projektartspezifisch die Zusammenhänge zwischen Function Points und Aufwendungen hergeleitet werden.

Es gibt unterschiedliche Zählregeln für Function Points. In diesem Text werden die Regeln der „International Function Point Users Group“ (IFPUG 1994) verwendet. In IBM (1983) und Seibt (1987) werden andere Gewichtungskriterien und nur sieben Einflussfaktoren zur Berechnung des Wertkorrekturfaktors herangezogen. Noth und Kretzschmar (1984) verwenden wie IFPUG vierzehn Einflussfaktoren, gewichten aber den Faktor 9 (technische Komplexität) mit 0-50 statt nur mit 0-5. Symons (1988) kritisiert das Albrechtsche Zählverfahren und schlägt

ein Zählverfahren vor, das statt kompletter Ein- und Ausgben die einzelnen Datenfelder zählt (Mark II Function Points).

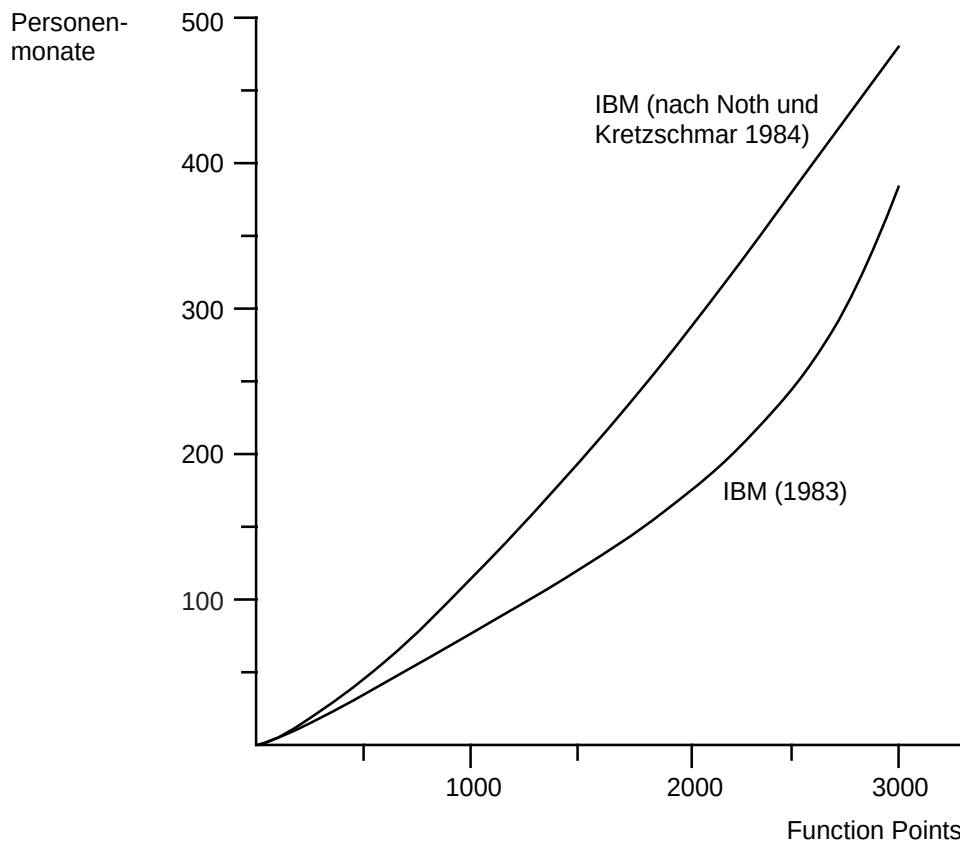


BILD 5.6. Bestimmung des Aufwand aus den Function Points

Jones (1995) gibt Erfahrungswerte für die Umrechnung von Function Points in Codezeilen an (Tabelle 5.4). Die Codezeilen verstehen sich ohne Leerzeilen und ohne Kommentar.

TABELLE 5.4. Anzahl Codezeilen pro Function Point

Sprache	Mittlere Anzahl Codezeilen
Assembler	320
C	128
FORTRAN	107
COBOL	197
Pascal	91
C++	53
Ada 95	49
Smalltalk	21
SQL	12

Das Function Point-Verfahren hat den großen Vorteil, dass die benötigten Eingangsgrößen sich in den frühen Phasen eines Projekts leichter bestimmen lassen als beispielsweise die Größe des erwarteten Results bei COCOMO. Zudem sind mit Function Points Produktivitätsmessungen möglich, die weniger leicht verfälschbar sind als Messungen auf der Grundlage der erzeugten Programmzeilen.

Allerdings ist das Verfahren auf Informationssysteme zugeschnitten und daher auf andere Arten von Software (zum Beispiel Prozessautomatisierung) nur beschränkt anwendbar. Ein wei-

terer Nachteil ist, dass das Function Point-Maß nicht additiv ist: Die Summe der Function Points von n logisch zusammenhängenden Teilsystemen ist größer als die Anzahl der Function Points des Gesamtsystems. Dies liegt daran, dass die Schnittstellen zwischen je zwei Teilsystemen auf beiden Seiten als Schnittstellen zu externen Datenbeständen gezählt werden, während diese Daten bei Betrachtung des Gesamtsystems als *ein* interner Datenbestand gezählt werden. Es ist daher nicht möglich, bei einem großen System die Function Points pro Teilsystem zu bestimmen und diese zu addieren.

5.5 Sonstige Methoden

Es gibt eine Reihe weiterer gängiger Methoden zur Kostenschätzung, z.B. so schätzen, dass man auf jeden Fall den Auftrag bekommt, so viel schätzen, wie der Auftraggeber zu zahlen bereit ist, Schätzung nach dem Parkinson'schen Gesetz: „Das Projekt kostet soviel Arbeitskapazität wie vorhanden ist“, usw.

5.6 Schätzung von Pflegekosten

Aus Messungen über Entwicklungs- und Pflegekosten von Software ergeben sich die folgenden beiden Faustregeln für Pflegekosten.

REGEL 5.2. Das Kostenverhältnis zwischen Entwicklung und Pflege eines Software-Produkts liegt im Bereich von 30:70 bis 50:50. Je länger ein Produkt lebt und je schlechter seine Qualität ist, desto höher ist der Kostenanteil für die Pflege.

REGEL 5.3. Die Kosten für die Pflege eines Software-Produkts verteilen sich etwa wie folgt: 60% Verbesserungen, 20% Anpassungen und 20% Fehlerbehebung.

Bild 5.7 zeigt verschiedene Schätzwerte für die Anzahl Codezeilen, die ein Vollzeit-Pflegeprogrammierer betreuen kann. Die Aussage 20 KDSI/FSP_M¹ bedeutet beispielsweise, dass zur Pflege eines Programms mit 20000 Codezeilen die volle Arbeitskraft einer Person (über das ganze Jahr hinweg) erforderlich ist.

Source	Application Type	(KDSI/FSP) _M
COCOMO, lowest		3
[Wolverton, 1980]	Aerospace	8
[Förrens-Harris, 1979]	Aerospace	10
COCOMO, 25th percentile		10
[Daly, 1977]	Real-time	10–30
[Griffin, 1980]	Real-time	12
[Elliott, 1977]	Business	20
[Graver and others, 1977]	Business, HOL	20
[Graver and others, 1977]	Business, MOL	22
COCOMO, median		25
[Lientz-Swanson, 1980]	Business, 487 installations	32
COCOMO, 75th percentile		36
[Daly, 1977]	Support software	30–120
COCOMO, highest		132

Quelle: Boehm (1981)

BILD 5.7. Zeile/Person-Raten in der Software-Pflege

¹ KDSI/FSP_M steht für kilo delivered source instructions per full-time software person (for maintenance).

Sind die Function Points der gepflegten Systeme bekannt und werden die Pflegeaufwendungen über längere Zeit erfasst, so kann daraus der mittlere Pflegeaufwand pro Function Point für das betreffende Unternehmen bestimmt werden. Eine solche Zahl kann dann auch für Prognosen über zu erwartende Pflegeaufwendungen eines neuen Systems verwendet werden. Jones (1996) gibt dazu folgende Faustregel an.

REGEL 5.4. *Faustregel von Jones für Pflegeaufwand.* Zur Pflege eines Systems von N Function Points sind $N/500$ Personen erforderlich.

5.7 Einfluss der Schätzung auf den Aufwand

Es gibt Hinweise dafür, dass Schätzung und tatsächlicher Aufwand keine voneinander unabhängigen Größen sind. Dies ist das durch das Gesetz von Parkinson (Der Aufwand passt sich der verfügbaren Zeit an) beschriebene Phänomen (Bild 5.8). Abdel-Hamid und Madnick (1986) haben in Simulationen eine signifikante Korrelation zwischen der Schätzung und dem tatsächlichen Aufwand gefunden. Diese Ergebnisse dürfen allerdings nicht zu falschen Schlüssen verleiten. Das Parkinson-Phänomen tritt vor allem dann auf, wenn in zu knapp kalkulierten Terminplänen Luft geschaffen wird. Dann werden nämlich all die Arbeiten tatsächlich gemacht, die bei ordnungsgemäßer Entwicklung eigentlich gemacht werden sollten (z.B. sorgfältige Dokumentation und Prüfung), die aber sonst der Termin-Guillotine zum Opfer fallen. In Bild 5.8 beispielsweise beeinflusst die Schätzung ab etwa dem 1,6 fachen des ursprünglichen Werts den tatsächlichen Aufwand kaum noch.

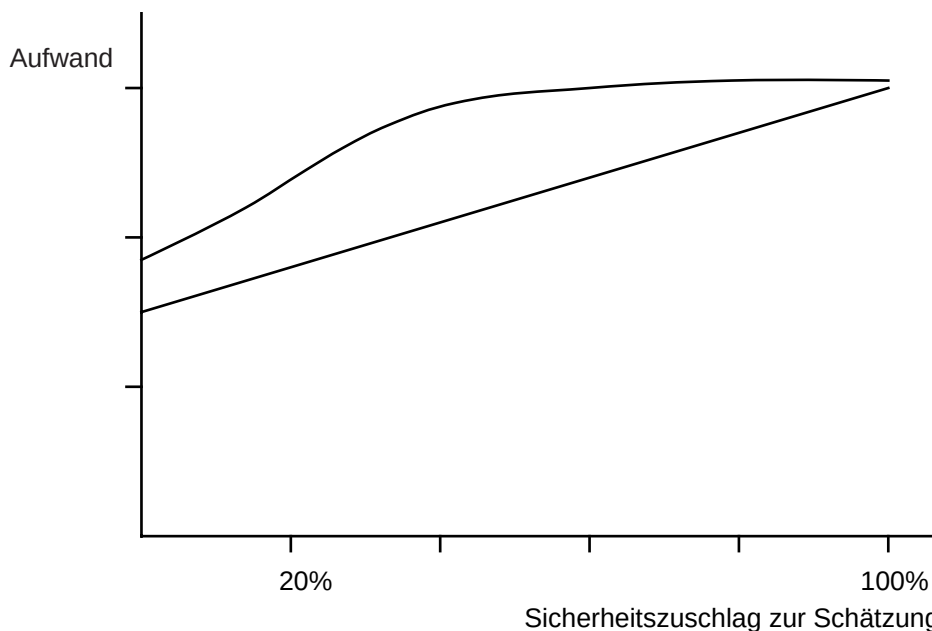


BILD 5.8. Der Parkinson-Effekt: Korrelation von geschätztem und effektivem Aufwand

Aufgaben

- 5.1 Sie sind Mitarbeiter(in) in der Informatikabteilung eines Unternehmens. Im Rahmen des unternehmensweiten Qualitätsmanagementsystems nehmen Sie in Ihrer Abteilung die Rolle des Qualitätsbeauftragten ein.
- Ihr Chef hat auf einer Konferenz von COCOMO gehört. Da es mit der Aufwandschätzung in Ihrer Abteilung im Argen liegt, hat er beschlossen, das Gehörte in die Tat umzusetzen und ab sofort alle den Aufwand für alle neuen Projekte mit COCOMO schätzen zu lassen. Bevor er seinen Entschluss verkündet, fragt er Sie als Qualitätsbeauftragte(n) noch um Ihre Meinung. Was raten Sie ihm?
- 5.2 Im Rahmen der Entwicklung eines Informationssystems für die Teilnehmeradministration eines Schulungsunternehmens ist eine Anwendung zur Anmeldung von Kursteilnehmern zu erstellen. Die Anwendung wird über untenstehende Eingabemaske angesprochen. Unterstrichene Felder sind Eingabefelder, graue Felder sind Ausgabefelder, Schaltknöpfe lösen Verarbeitungen und Datenbankzugriffe aus. Die Anwendung verwendet die Gegenstandstypen KURS, KUNDE und BUCHUNG aus der Datenbank.

Kursanmeldung

Kurs-Nr
Kurs

Daten

Teilnehmer aktuell maximal

Anzumeldende Person ☒ Neuer Kunde
☐ bekannter Kunde KD-Nr

Name

Titel

Vorname

Strasse, Nr.

/ Postfach

PLZ Ort

Berechnen Sie die Function Points für diese Anwendung. Beachten Sie folgende Hinweise:

- Die Maske enthält neben der eigentlichen Transaktion (Anmeldung) zwei weitere eingebettete Transaktionen (Kurs suchen, Person suchen). Diese sind mitzuzählen, wenn sie nicht anderswo bereits als selbständige Transaktionen existieren. Gehen Sie davon aus, dass das hier der Fall ist, d.h. dass beide eingebetteten Transaktionen gezählt werden müssen.
- Anfragen, die nicht mehr als einen Datenbestand betreffen und nicht mehr als 15 Ein- oder Ausgabedaten umfassen, werden als „einfach“ bewertet.
- Sie können davon ausgehen, dass die beteiligten Datenbestände alle als „einfach“ zu bewerten sind.
- Bei den Einflussfaktoren können Sie von folgenden Annahmen ausgehen: Belastung der Hardware: unbedeutend, komplexe Verarbeitungen: nicht vorhanden, Wiederverwendbarkeit: sehr wichtig. Alle übrigen Faktoren haben durchschnittlichen Einfluss.

Ergänzende und vertiefende Literatur

Boehm (1981) ist die Standardreferenz für COCOMO.

IFPUG (1994) enthält detaillierte Zählregeln für Function Points und illustriert diese anhand von Beispielen.

Knöll und Busse (1991) beschreiben verschiedene, in der Praxis eingesetzte Verfahren, u.a. auch COCOMO und Function Points.

Wellman (1992) und Jones (1998) sind Monographien über Software-Kosten; letztere präsentiert den Stand der Technik in umfassender Weise.

Zitierte Literatur

Abdel-Hamid, T.K., S.E. Madnick (1986). Impact of Schedule Estimation on Software Project Behavior. *IEEE Software* **3**, 4 (July 1986). 70-75.

Albrecht, A.J. (1979). Measuring Application Development Productivity. *Proceedings Guide/Share Application Development Symposium* (Oct. 1979). 83-92

Boehm, B. (1981). *Software Engineering Economics*. Englewood Cliffs, N.J.: Prentice Hall.

IBM (1983). *Die Function Point-Methode: Eine Schätzmethode für IS-Anwendungs-Projekte*. Broschüre GE 12-1618-0. IBM Deutschland GmbH.

IFPUG (1994). *Function Point Counting Practices Manual, Release 4.0*. International Function Point Users Group. Westerville, Ohio (USA).

Jones, T.C. (1995). Backfiring. Converting Lines of Code to Function Points. *IEEE Computer* **28**, 11 (Nov 1995). 87-88.

Jones, T.C. (1996). Software Estimating Rules of Thumb. *IEEE Computer* **29**, 3 (March 1996). 116-118.

Jones, T. C. (1998). *Estimating Software Costs*. New York: McGraw-Hill.

Knöll, H.-D., J. Busse (1991). *Aufwandschätzung von Software-Projekten in der Praxis*. Reihe Angewandte Informatik Band 8, Mannheim, Wien Zürich: BI-Wissenschaftsverlag.

Noth, T. und M. Kretzschmar (1984). *Aufwandschätzung von DV-Projekten*. Berlin, etc.: Springer.

Seibt, D. (1987). Die Function-Point-Methode: Vorgehensweise, Einsatzbedingungen und Anwendungserfahrungen. *Angewandte Informatik* 1/1987. 3-11

Symons, C.R. (1988). Function Point Analysis: Difficulties and Improvements. *IEEE Transactions on Software Engineering* **14**, 1 (Jan. 1988). 2-11.

Wellman, F. (1992). *Software Costing*. Hemel Hempstead: Prentice Hall International (UK).

6. Konzipieren von Lösungen

6.1 Motivation

Für die Entwicklung von Kleinsoftware (typisch durch eine Person für den Eigengebrauch) ist kein systematischer Entwurf notwendig. Allenfalls werden Skizzen gemacht, im Übrigen wird direkt programmiert.

Größere Software dagegen braucht einen systematischen, strukturierten Aufbau, der nur durch ein sorgfältig erstelltes Lösungskonzept sichergestellt werden kann. Ohne ein solches Lösungskonzept ist es nicht oder nur mit großem Aufwand möglich

- die Lösung zu verstehen,
- die Entwicklungsarbeit auf mehrere Personen zu verteilen (Software-Entwicklung ist Teamarbeit!),
- die Lösung in vorhandene Software einzubetten (dazu müssen die Schnittstellen und Wechselwirkungen verstanden werden),
- die Lösung zu verteilen (auf mehrere Rechner, ggf. auch geographisch).

Ferner legt ein gutes Lösungskonzept den Grundstein für ein leicht pflegbares System. Es erleichtert das Auffinden zu ändernder Teile, ermöglicht lokale Änderungen unter Wahrung der ursprünglichen Strukturen und erlaubt, die Auswirkungen von Erweiterungen und Ergänzungen abzuschätzen und zu begrenzen.

In der Konzipierung begangene Fehler sind häufig gar nicht oder mit nur großem Aufwand reparierbar. Soll Software beschafft werden, so muss in der Konzipierung dafür gesorgt werden, dass Aufgabe und Lösung hinreichend zusammenpassen. Sorgfältiges Konzipieren ist daher für jede Art von professionell entwickelter Software wirtschaftlich.

6.2 Definitionen und Begriffe

Das *Konzipieren einer Lösung* ist die Erstellung und Dokumentation des *Architekturentwurfs* oder Grobentwurfs eines Systems. Dabei werden die wesentlichen Komponenten der Lösung und die Interaktionen zwischen diesen Komponenten festgelegt. Im Englischen spricht man von *Architectural Design*.

DEFINITION 6.1. *Architektur (architecture).* Die Organisationsstruktur eines Systems oder einer Komponente (IEEE 610.12).

DEFINITION 6.2. *Entwurf (design).* 1. Der Prozess des Definierens von Architektur, Komponenten, Schnittstellen und anderen Charakteristika eines Systems oder einer Komponente. 2. Das Ergebnis des Prozesses gemäß 1. (IEEE 610.12).

DEFINITION 6.3. *Lösungskonzept.* Das Dokument, welches das Konzept der Lösung, d.h. die Architektur der zu erstellenden Software dokumentiert. Synonyme: *Software-Architektur*, *Systemarchitektur*, *software architecture*, *system architecture*.

Hinweis: Wenn von «System» die Rede ist, kann sowohl die zu erstellende Software als auch die technische bzw. organisatorische Umgebung, in welche die Software eingebettet ist, gemeint sein. Die Unterscheidung ergibt sich aus dem Kontext oder durch explizites Rückfragen. In diesem Kapitel meint «System» immer die zu erstellende Software, sofern nicht ausdrücklich etwas anderes gesagt wird.

6.3 Entwurfsprinzipien

6.3.1 Strukturen und Abstraktionen

Ein systematischer Aufbau ist die Grundlage jedes guten Entwurfs. Die Systematik besteht in der Wahl bzw. Verwendung geeigneter Strukturen und Abstraktionen. Die *Strukturen* gliedern das Lösungskonzept in *Komponenten* (Module, Prozesse, externe Akteure,...) und *Interaktionen* zwischen Komponenten (Aufrufe, Beziehungen, Nachrichten,...). Die *Abstraktionen* unterstützen das Verstehen des Konzepts durch systematische Vergröberung und Verfeinerung der Darstellung nach verschiedenen Kriterien. Sie ermöglichen dabei

- das Sichtbarmachen großer Zusammenhänge unter Weglassung der Details
- die Darstellung eines Details unter Weglassung/starker Vergröberung des Rests
- die Herstellung eines systematischen Zusammenhangs zwischen Übersichten und Detailsichten.

In der Konzipierung werden hauptsächlich vier Abstraktionsarten verwendet: Komposition, Benutzung, Generalisierung, Aspekte.

- Die hierarchische Zerlegung von komplexen Lösungen in gekapselte, in sich geschlossene Teillösungen (siehe Abschnitt 6.3.2) verwendet die *Kompositionsabstraktion*.
- Die Verwendung von Komponenten ohne Kenntnis ihres inneren Aufbaus (siehe Abschnitte 6.3.3, 6.6.2 und Bild 6.3) ist eine Anwendung der *Benutzungsabstraktion*.
- Die Bildung von Klassenhierarchien im objektorientierten Entwurf (vgl. Abschnitt 6.6.3), bei der jedes Objekt einer Unterklasse einen Spezialfall von Objekten aller ihrer Oberklassen darstellt, basiert auf der *Spezialisierungsabstraktion*.
- Die separate Darstellung von Querschnittsaufgaben (vgl. Abschnitt 6.3.7) ist ein Beispiel für die Verwendung der *Aspektabstraktion*.

6.3.2 Modularität

Die Modularisierung ist eine der Hauptaufgaben beim Konzipieren von Lösungen. Sie gliedert eine Lösung in kleinere, besser verstehbare und überschaubare Teillösungen. Module können rekursiv wieder in Module zerlegt werden, bis die Teillösungen klein genug sind. Modularisierung ist *das* Entwurfsprinzip schlechthin. Ohne eine gute Modularisierung ist die Komplexität großer Systeme nicht mehr beherrschbar.

DEFINITION 6.4. *Modul (module).* Eine benannte, klar abgegrenzte Komponente eines Systems.

Bei der Modularisierung kommt es wesentlich darauf an, dass nicht irgendwie zerlegt wird, sondern so, dass die Module in sich geschlossene Teillösungen mit möglichst wenigen, wohldefinierten Interaktionen zu anderen Modulen bilden. Wir verlangen von einer guten Modularisierung, dass sie folgendes leistet:

- Jeder Modul ist eine in sich geschlossene Einheit
- die Verwendung des Moduls erfordert keine Kenntnisse über seinen inneren Aufbau

- die Kommunikation eines Moduls mit seiner Umgebung erfolgt ausschließlich über eine klar definierte Schnittstelle
- Änderungen im Inneren eines Moduls, welche seine Schnittstelle unverändert lassen, haben keine Rückwirkungen auf das übrige System
- die Korrektheit des Moduls ist ohne Kenntnis seiner Einbettung ins Gesamtsystem nachprüfbar.

Die *Güte* einer Modularisierung lässt sich im Wesentlichen an folgenden drei Kriterien messen: den Maßen Kohäsion und Kopplung (Stevens, Myers und Constantine, 1974) sowie der geeigneten Berücksichtigung von Ressourcen (vgl. Abschnitt 6.3.6).

DEFINITION 6.5. Kohäsion (cohesion). Ein Maß für die Stärke des inneren Zusammenhangs eines Moduls. Je höher die Kohäsion ist, desto besser.

Module mit *zufälliger* Kohäsion (bunt zusammengewürfelte Teile) oder *zeitlicher* Kohäsion (zusammengefasst ist, was zeitlich miteinander auszuführen ist) sind softwaretechnisch schlecht und sollten vermieden werden. Anzustreben sind *funktionale* Kohäsion (der Modul realisiert eine in sich geschlossene Funktion) oder *objektbezogene* Kohäsion (der Modul enthält ein Datenobjekt und alle zu seiner Bearbeitung erforderlichen Operationen). Dazwischen gibt es verschiedene Kohäsionsstufen (*logisch, kommunikativ, sequentiell, prozedural*), die hier nicht näher diskutiert werden sollen. Für Details sei auf Stevens, Myers und Constantine (1974) sowie auf Page-Jones (1988) verwiesen.

DEFINITION 6.6. Kopplung (coupling). Ein Maß für die Abhängigkeit zwischen zwei Modulen. Die Modularisierung ist umso besser, je geringer die wechselseitige Kopplung zwischen den Modulen ist.

Inhaltskopplung (ein Modul verändert direkt lokale Daten oder Code eines anderen Moduls) ist unter allen Umständen zu vermeiden. *Globale* Kopplung (Daten, die von allen Modulen gelesen und verändert werden können) ist zu vermeiden oder allenfalls auf wenige Daten zu beschränken. *Datenbereichskopplung* (Module kommunizieren über Datenbereiche, die sie gemeinsam haben, oder sich gegenseitig zur Verfügung stellen) ist akzeptabel.

Anzustreben ist *Datenkopplung* (nur die tatsächlich benötigten Daten werden über wohldefinierte Schnittstellen übertragen).

Auch die konsequente Anwendung des Prinzips, dass jedes Datum einem Modul gehört und nur von diesem verändert werden darf, reduziert die Kopplung. Bei Bedarf stellt der Eigentümer eines Datums Operationen zur kontrollierten Manipulation des Datums durch Dritte bereit.

6.3.3 Das Geheimnisprinzip (Information Hiding): Das Fundamentalprinzip zur Gliederung komplexer Systeme

Das von Parnas (1972) entdeckte Geheimnisprinzip (information hiding) ist das Gliederungskriterium, welches gute Modularisierungen mit den oben genannten Eigenschaften liefert.

DEFINITION 6.7. Geheimnisprinzip (information hiding). Kriterium zur Gliederung eines Gebildes in Komponenten, so dass jede Komponente eine Leistung (oder eine Gruppe logisch eng zusammenhängender Leistungen) vollständig erbringt und zwar so, dass außerhalb der Komponente nur bekannt ist, was die Komponente leistet. Wie sie ihre Leistungen erbringt, wird nach außen *verborgen*.

Parnas hat das Geheimnisprinzip ursprünglich definiert als das Einkapseln von Entwurfsentscheidungen beim Modularisieren von Software mit dem Ziel, die *Realisierung* der Entwurfsentscheidungen vor den Modulverwendern zu *verbergen* und die Entwürfe dadurch leichter

änderbar zu machen. Die Definition 6.7 wurde bewusst allgemeiner gefasst, um deutlich zu machen, dass es sich hier um ein weit über den Software-Entwurf und die Informatik hinausreichendes, *allgemeines und fundamentales Abstraktionsprinzip zur Beherrschung komplexer Systeme* handelt.

Wir benutzen das Geheimnisprinzip tagtäglich zur Beherrschung der Komplexität unserer Umwelt; etwa indem wir Auto fahren, ohne wissen zu müssen, wie ein Verbrennungsmotor funktioniert oder ein elektrisches Gerät in Betrieb nehmen, ohne wissen zu müssen, wie der Strom in die Steckdose kommt.

Eine Bank ist aus der Sicht der Kunden ebenfalls ein klassisches nach dem Geheimnisprinzip arbeitendes System: Sie erbringt für ihre Kunden über eine Schnittstelle (Schalter, Bancomat) Leistungen (Geld entgegennehmen und sicher aufbewahren, Guthaben wieder auszahlen,...). Dabei wissen die Kunden nur, *was* die Bank leistet. *Wie* die Leistungen erbracht werden, zum Beispiel wie die Bank die Kontenführung organisiert oder wie sie die Kundeneinlagen so anlegt, dass sie verlangte Auszahlungen jederzeit leisten kann, bleibt nach außen verborgen. Das Geheimnisprinzip ist für beide Seiten wichtig: Die Kunden wollen ihr Konto benutzen, ohne wissen zu müssen, wie eine Bank funktioniert und ohne von bankinternen Organisationsänderungen betroffen zu sein. Die Bank will ihr Geschäft unabhängig von den Kunden organisieren, und sie will insbesondere Änderungen in der Realisierung ihrer Leistungen vornehmen können, ohne dies mit allen Kunden abstimmen zu müssen.

Im Software-Entwurf sind die Komponenten Module, die Leistungen Teillösungen. Das WAS ist in der Modulschnittstelle beschrieben, das WIE sind die Entwurfsentscheidungen und ihre Realisierung durch Programme. Bei Software gibt es vier typische Arten von Entwurfsentscheidungen:

- wie eine Funktion realisiert ist, zum Beispiel die Kalkulation einer Versicherungsprämie
- wie ein Objekt aus dem Anwendungsbereich repräsentiert und realisiert ist, zum Beispiel ein Konto
- wie eine Datenstruktur aufgebaut ist und wie sie bearbeitet werden kann, zum Beispiel die Datenhaltung und die Suchfunktion für die Bereitstellung von «Gelben Seiten» im WWW
- wie Leistungen Dritter realisiert sind, zum Beispiel wie ein Datenbanksystem intern funktioniert.

6.3.4 Schnittstellen und Verträge

Schnittstellen (interfaces) beschreiben das Leistungsangebot eines Moduls. In Anwendung des Geheimnisprinzips soll die Beschreibung so weit wie möglich realisierungsneutral sein. In der Regel werden Leistungen in Form von Operationen und Datentypen bereitgestellt. Die Bereitstellung direkt zugreifbarer Daten wird nach Möglichkeit vermieden, da sie die Offenlegung der Datenrepräsentation erfordert. Voraussetzungen (preconditions) und Ergebniszusicherungen (assertions, postconditions) gestatten eine realisierungsneutrale Beschreibung der Operationen:

- Die *Voraussetzungen* geben an, welche Bedingungen erfüllt sein müssen, bevor die Operation aktiviert werden darf. Die Operation prüft die von ihr gemachten Voraussetzungen in der Regel nicht. Bei verletzten Voraussetzungen ist das Ergebnis der Operation undefiniert.
- Die *Ergebniszusicherungen* beschreiben die Ergebnisse der Operation (gelieferte Daten, Änderungen im Systemzustand)

Weitere Aussagen, sei es für eine Operation oder für den ganzen Modul, können mit Invarianten (invariants) und Verpflichtungen (obligations) gemacht werden:

- *Invarianten* sind Zusicherungen über Systemeigenschaften, die durch die Operation nicht verändert werden
- *Verpflichtungen* überbinden dem Verwender einer Operation Pflichten, die er erfüllen muss, um übergeordnete Systeminvarianten nicht zu verletzen.

Die Bedingungen und Zusicherungen beschreiben einen *Vertrag* zwischen dem Modul als Leistungserbringer und den Klienten, welche die Leistungen durch Verwendung von Operationen des Moduls nutzen: Unter der Bedingung, dass der Klient garantiert, dass die Voraussetzungen einer Operation des Moduls zum Zeitpunkt der Aktivierung erfüllt sind, garantiert der Modul, dass die Ergebniszusicherungen der Operation nach ihrem Abschluss erfüllt sind. Die konsequente Dokumentation aller Modulschnittstellen durch solche Verträge trägt erheblich zur Robustheit und Änderbarkeit der Software bei. Meyer (1988/1997 und 1992) hat dafür den Ausdruck «Design by Contract» geprägt.

6.3.5 Nebenläufigkeit

Dort wo es möglich ist, werden Informatikprobleme durch *sequentielle* Programme realisiert. Unter sequentiellen Programmen verstehen wir solche, die ein Problem durch eine sequentielle Folge von Programmschritten lösen. Zu jedem Zeitpunkt ist folglich höchstens ein Programmschritt in Bearbeitung. Die Bevorzugung sequentieller Programme hat vor allem zwei Gründe: Erstens sind sequentielle Programme auf klassischen Rechnern leicht realisierbar und zweitens sind sie leichter zu verstehen und zu entwerfen als Programme, in denen mehrere Dinge gleichzeitig parallel ablaufen.

Viele Probleme erfordern jedoch die gleichzeitige, koordinierte Bearbeitung mehrerer Aufgaben, sei es lokal an einem Ort oder geographisch verteilt auf mehrere Orte. Solche Probleme sind nicht oder nur sehr schlecht mit sequentiellen Programmen lösbar. Stattdessen wird die Lösung in diesem Fall auf mehrere *nebenläufige Prozesse* aufgeteilt.

DEFINITION 6.8. *Prozess (process).* Eine durch ein Programm gegebene Folge von Aktionen, die sich in Bearbeitung befindet.

Ein Prozess entsteht also zu dem Zeitpunkt, wo seine erste Aktion begonnen wird und er verschwindet nach Abschluss seiner letzten Aktion. In dieser Zeit ist er entweder aktiv (d.h. er arbeitet nach dem gegebenen Programm) oder er wartet auf benötigte Ressourcen oder er wartet auf einen anderen Prozess (zum Beispiel um mit ihm zu kommunizieren).

DEFINITION 6.9. *Nebenläufigkeit (concurrency).* Die parallele oder zeitlich verzahnte Bearbeitung mehrerer Aufgaben.

Während ihrer Arbeit müssen Prozesse in der Regel miteinander *kommunizieren*, d.h. sie müssen *Informationen austauschen* oder ihren Arbeitsfortschritt *synchronisieren*. Häufig wird auch beides miteinander kombiniert.

Die zur Synchronisation notwendigen Konzepte und Konstrukte werden vom Betriebssystem oder von einer Laufzeitumgebung für die verwendete Programmiersprache bereitgestellt. Interessierte Leserinnen und Leser seien auf die Literatur über Betriebssysteme verwiesen.

Informationsaustausch erfolgt entweder über das Senden und Empfangen von Nachrichten oder über Speicherbereiche, die beiden kommunizierenden Prozessen zugänglich sind. Nachrichtenaustausch ist etwas langsamer, dafür aber sicher und geographisch verteilbar. Kommunikation über gemeinsame Speicher ist schnell, muss dafür aber explizit gesichert werden und ist in der Regel nicht verteilbar.

Beispiel

Ein Informatiksystem soll Devisenhändler bei ihrer Arbeit unterstützen. Zu diesem Zweck sollen auf einer Arbeitsplatzstation folgende Komponenten verfügbar sein:

- ein Editor zur Bearbeitung und Anzeige der laufenden Kauf- und Verkaufsaufträge,

- ein Archivierer, der alle abgewickelten Transaktionen registriert und bei Bedarf anzeigt,
- ein Telefonbutler, der ein Telefonverzeichnis anzeigt, bei Anwahl eines Eintrags die zugehörige Person anruft sowie mehrere Linien gleichzeitig hält und makelt,
- ein Kursbeobachter, der die aktuellen Kurse und den Kursverlauf der gehandelten Währungen fortlaufend ermittelt und anzeigt.

Alle vier Komponenten sollen gleichzeitig arbeiten können und werden daher mit nebenläufigen Prozessen realisiert. Für gemeinsame Infrastrukturaufgaben (zum Beispiel Datenhaltung) werden weitere Prozesse definiert.

6.3.6 Berücksichtigung der Ressourcen

Eine (zumindest vorläufige) *Zuordnung von Ressourcen* zu den Komponenten der Software-Architektur ist bereits in der Konzipierung notwendig, weil sonst die technische Machbarkeit des Lösungskonzepts und die Erfüllbarkeit der gestellten Anforderungen (insbesondere der Leistungsanforderungen) nicht überprüft werden können. Ressourcen können sowohl logischer Art (z.B. Prozesse) als auch physischer Art (z.B. Prozessoren) sein. Folgende Zuordnungen müssen vorgenommen werden:

- Module zu Prozessen
- Prozesse zu Prozessoren
- Daten zu Speichern bzw. Medien
- Prozesskommunikation zu Kommunikationstechnologien und -medien.

Die wesentlichen Kriterien dabei sind (in dieser Reihenfolge): Erfüllung der Anforderungen, Klarheit der Struktur und Ressourceneffizienz. Angestrebt wird eine möglichst hohe Übereinstimmung zwischen der logischen und der physischen Struktur des Systems. Solange die Leistungsanforderungen erfüllt sind, werden dafür auch Ineffizienzen, z.B. ungleichmäßig ausgelastete Prozessoren, in Kauf genommen. Eine test- und änderungsfreundliche Software-Struktur ist besser und wirtschaftlicher als eine chaotische Struktur mit optimal ausgelasteten Ressourcen.

6.3.7 Aspektorientierung

Während man insgesamt versucht, die Lösung in voneinander möglichst unabhängige Teillösungen zu gliedern, gibt es Querschnittsaufgaben, die nur in einer Gesamtsicht befriedigend konzipiert werden können. Dabei wird die Darstellung auf den betreffenden Querschnittsaspekt fokussiert. Ein Gestaltungskonzept für die Benutzerschnittstelle zum Beispiel muss systemweit gelten. Es wäre schlecht, in verschiedenen Modulen unterschiedliche Benutzerschnittstellen zu haben. Für die folgenden Querschnittsaufgaben sollten aspektorientierte Teilkonzepte erstellt werden, sofern dieser Aspekt für das betreffende System relevant ist: Die Datenhaltung, insbesondere das konzeptionelle Datenbankschema bei Verwendung einer Datenbank, die Gestaltung der Benutzerschnittstelle, die Behandlung von Fehlern sowie die Gewährleistung von Sicherheit und Fehlertoleranz.

6.3.8 Nutzung von Vorhandenem

Wo immer möglich, werden Systeme nicht vollständig neu entwickelt, sondern es werden Teile oder gar das ganze System *beschafft* bzw. *wiederverwendet*. In der Konzipierung müssen die wesentlichen Beschaffungs- und Wiederverwendungsentscheidungen getroffen werden. Es ist daher eine zentrale Aufgabe der Konzipierung, zu untersuchen, welche Teile der Lösung mit welchen existierenden Komponenten realisiert werden können. Folgende Punkte sind immer zu untersuchen:

- Kann eine Lösung vollständig beschafft werden (Standardsoftware, konfigurierbare Bausteine)?
- Können abgeschlossene Teilsysteme (zum Beispiel ein Datenbanksystem) beschafft werden?
- Ist das System durch Einbettung in einen existierenden Software-Rahmen (framework) realisierbar?
- Können einzelne Komponenten (aus Programm- oder Klassenbibliotheken) genutzt werden?
- Können Architektur- und Entwurfsideen wiederverwendet werden (zum Beispiel durch Verwendung vorhandener Architekturmetaphern und -muster oder von Entwurfsmustern (design patterns, Gamma et al. 1995)?
- Kann/soll das Lösungskonzept so modifiziert werden, dass vorhandene Software wiederverwendet bzw. beschafft werden kann?

Für Beschaffungen und Wiederverwendung gelten die gleichen Abwägungen und Kostenüberlegungen wie für Entwurfsvarianten (vgl. Abschnitt 6.5.4). Bei den Kostenüberlegungen sind zwei spezifische Punkte zu beachten:

- Wie und in welchem Zeit- und Kostenrahmen können Kandidaten gesucht und evaluiert werden?
- Welche Parametrierungen, Konfigurierungen, Anpassungen und Ergänzungen sind notwendig und wieviel kosten diese?

6.3.9 Ästhetik

Wie in der Architektur von Gebäuden gibt es auch in der Software-Architektur eine Ästhetik. Diese äußert sich vor allem in

- der Wahl und konsequenten Verwendung eines Architekturstils (vgl. Abschnitt 6.6)
- klar erkennbaren, gestalteten Strukturen (dies im Gegensatz zu irgendwie Gewordenem oder Gewurstem)
- einer der Struktur des Problems angemessenen Struktur der Architektur
- der Wahl der einfachsten und klarsten Lösung aus der Menge der möglichen Lösungen.

6.3.10 Qualität

Ein guter Entwurf ist:

- *effektiv*, das heißt er erfüllt die Vorgaben und löst das Problem des Auftraggebers,
- *wirtschaftlich*, das heißt gebrauchstauglich, kostengünstig und mehrfachverwendbar bzw. mehrfachverwendet,
- *softwaretechnisch gut*, das heißt leicht verständlich, robust, zuverlässig und änderungsfreundlich.

Die Sicherstellung dieser Qualitäten erfordert kontinuierliche Prüfmaßnahmen im Entwurfsprozess (vgl. Abschnitt 6.5.3, Prüfung).

6.4 Produkte

Das Ergebnis des Architekturentwurfs wird in einem Dokument niedergelegt, das wir *Lösungskonzept* oder auch *Software-Architektur* nennen. Das Lösungskonzept dokumentiert die Gliederung der Software in Prozesse, die Modularisierung, die Interaktionen zwischen den Prozessen und Modulen, die Zuordnung von Ressourcen sowie aspektbezogene Teilkonzepte für Querschnittsaufgaben wie Datenbankschema, Konzept der Mensch-Maschine-Kommunikation,

Fehlertoleranzkonzept, etc. Das Lösungskonzept dokumentiert auch die Einbettung in vorhandene Software sowie Beschaffungs- und Wiederverwendungsentscheide.

MUSTER 6.1. Möglicher Aufbau eines Lösungskonzepts

1. Einleitung

1.1 Überblick

Überblick über die gewählte Lösung

1.2 Ziele und Vorgaben

Beschreibung von Entwurfszielen und Vorgaben, die nicht in der Anforderungsspezifikation stehen

1.3 Einbettung und Abgrenzung

- Wo und wie ist das konzipierte System eingebettet
- Wie und über welche Schnittstellen wird mit der Umwelt kommuniziert

1.4 Lösungsalternativen

Betrachtete, aber schließlich verworfene Lösungsalternativen: Kurze Skizze jeder Alternative, Grund für die Verwerfung

2. Struktur der Lösung

2.1 Übersicht

- Architekturstil, Metapher(n) und Architekturmuster, die der Architektur zugrunde liegen
- Teilsysteme und ihre Aufgaben

2.2 Prozessstruktur

Prozesse und Kommunikation zwischen den Prozessen

2.3 Modulare Struktur

Module und ihre Zusammenhänge, bei objektorientiertem Entwurf Klassen- bzw. Objektmodelle

2.4 Entwurf der Module

- Beschreibung der Schnittstellen
- ggf. Hinweise zur geplanten Implementierung

2.5 Physische Struktur

- Physische Gliederung der Software in Pakete, Komponenten, etc.
- Ressourcenzuordnung

3. Aspektbezogene Teilkonzepte

Ein Unterkapitel je interessierendem Aspekt, zum Beispiel Datenhaltungskonzept, Mensch-Maschine-Kommunikationskonzept, Fehlerbehandlungskonzept, Fehlertoleranzkonzept, Sicherheitskonzept, etc.

4. Voraussetzungen und benötigte Hilfsmittel

4.1 Benötigte Software

Beschreibung der benötigten (fertigen) Software, welche für Entwicklung und/oder Betrieb des Systems zu beschaffen bzw. zu verwenden ist

4.2 Benötigte Hardware

Beschreibung der benötigten Hardware, welche für Entwicklung und/oder Betrieb des Systems zu beschaffen bzw. zu verwenden ist

4.3 Benötigtes Umfeld

- Charakterisierung der für den Betrieb des Systems erforderlichen organisatorischen und /oder technischen Strukturen und Abläufe

Quellennachweis

6.5 Der Entwurfsprozess

6.5.1 Einbettung

Die Erstellung des Architekturentwurfs ist der erste Schritt der Lösung eines Systems. Als Vorgabe muss eine Anforderungsspezifikation existieren. Je nach gewähltem Prozessmodell für den (gesamten) Entwicklungsprozess wird die Anforderungsspezifikation vollständig vor dem

Lösungskonzept erstellt oder die Erstellung erfolgt verzahnt, aber mit separater Dokumentation von Anforderungen einerseits und Lösung andererseits. Es sind sowohl hierarchische als auch zeitliche Verzahnung möglich. Bei der *hierarchischen Verzahnung* werden aus Globalanforderungen erste, grundsätzliche Lösungsentscheide abgeleitet. Diese induzieren Anforderungen an die Komponenten dieser Globallösung. Aus diesen Anforderungen werden wieder Lösungskonzepte abgeleitet, die wiederum zu Anforderungen an Teilsysteme dieser Lösungen führen können, usw. *Zeitliche Verzahnung* entsteht bei Verwendung von Wachstumsmodellen.

Das Lösungskonzept wiederum ist die Vorgabe für die Realisierung der Software, (d.h. den Detailentwurf, die Codierung und die Integration). Auch hier gibt es vor allem bei Wachstumsmodellen eine zeitliche Verzahnung von Architekturentwurf und Realisierung.

Der Architekturentwurf grenzt sich vom Detailentwurf ab, indem im Architekturentwurf nur die Komponenten, ihre Schnittstellen und ihre Interaktionen festgelegt werden, nicht aber die interne Realisierung der Komponenten durch Algorithmen und (verborgene) Datenstrukturen. Letzteres ist Aufgabe des Detailentwurfs.

6.5.2 Die Hauptaufgaben des Architekturentwurfs

Tabelle 6.1 fasst die Hauptaufgaben zusammen. Bild 6.1 zeigt ihre Zusammenhänge und Abhängigkeiten. Nachfolgend werden die einzelnen Aufgaben etwas näher charakterisiert. Das konkrete Vorgehen bei der Bearbeitung der Aufgaben ist stark von den verwendeten Entwurfstechniken und vom gewählten Prozessmodell abhängig.

TABELLE 6.1. Aufgaben des Architekturentwurfs

• Aufgabe analysieren • Anforderungen verstehen • Vorhandene bzw. beschaffbare Technologien und Mittel analysieren • Architektur modellieren und dokumentieren • Grundlegende Systemarchitektur festlegen: Wahl geeigneter Architekturmuster und -metaphern, Festlegung des Architekturstils • Modularisieren • Nebenläufige Lösungen in Prozesse gliedern • Wiederverwendungs- und Beschaffungsentscheide treffen • Ressourcen zuordnen • Aspektbezogene Teilkonzepte für Querschnittsaufgaben erstellen • Lösungskonzept (als Dokument) erstellen • Lösungskonzept prüfen • Anforderungen erfüllt? • Softwaretechnisch gut? • Wirtschaftlich?
--

6.5.3 Vorgehen beim Konzipieren

6.5.3.1 Aufgabenanalyse

Das Vorgehen bei der Aufgabenanalyse hängt sehr stark von der Art der vorhandenen Vorgaben ab. Teilweise können die gleichen Techniken wie bei der Gewinnung von Anforderungen (vgl. Kapitel 7) verwendet werden. Es geht darum, ein klares Verständnis zu erarbeiten, was verlangt wird und welche grundsätzlichen Lösungsmöglichkeiten in Frage kommen.

- Ist die Software auf mehrere Rechner verteilt, versucht man die Prozesse als Ganzes zu verteilen. Manchmal muss aber auch ein logischer Prozess auf mehrere physische Prozesse aufgeteilt werden.
- Weitere Prozesse können notwendig sein, wenn Aufgaben unterschiedlicher Dauer und Dringlichkeit zu erfüllen sind, wenn die Aufgaben, die ein Prozess zu erfüllen hat, zu umfangreich werden oder wenn Fehlertoleranzanforderungen zu erfüllen sind.

Die Kommunikationsbedürfnisse ergeben sich aus der Verteilung der Module auf die Prozesse.

Das Vorgehen bei *Wiederverwendung und Beschaffung* wird in Abschnitt 6.5.5 beschrieben.

Bei der *Ressourcenzuordnung* werden die Module in der Regel als Ganzes auf Prozesse verteilt. Je enger die Kopplung zwischen einer Gruppe von Modulen ist, desto eher sollten sie dem selben Prozess und dieser wiederum dem selben Rechner im System zugeordnet werden. Die Aufspaltung eines Moduls auf mehrere Prozesse oder gar mehrere Rechner erhöht den Kopplungsgrad des Systems und erschwert Fehlersuche und Modifikationen. Zu erfüllende Leistungsanforderungen (v.a. geforderte Reaktionszeiten und zu transportierende Datenvolumina) können aber zu Abweichungen von diesem Prinzip zwingen.

Die gleichen Überlegungen gelten für die Zuordnung von Prozessen und Daten. Bei der Verteilung auf mehrere Rechner ist immer der durch die Verteilung entstehende Kommunikationsbedarf, insbesondere die Menge der zu übertragenden Daten, zu berücksichtigen. Bei der Datenzuordnung muss dabei meistens zwischen redundanzfreier Speicherung mit erhöhtem Datenübertragungsbedarf einerseits und redundanter Speicherung mit geringem Übertragungsbedarf abgewogen werden.

Bei der Zuordnung der Prozesskommunikation muss vor allem darauf geachtet werden, dass die Leistungsanforderungen unter Nutzung der vorhandenen Technologie erreicht werden.

Dort, wo physische Randbedingungen oder Leistungsanforderungen dazu zwingen, eine von der logischen Struktur der Lösung abweichende physische Struktur zu wählen, muss neben der logischen auch die physische Struktur dokumentiert werden. Gleiches gilt, wenn die Software zu Liefer- oder Verwaltungszwecken in Pakete gegliedert wird. Das Lösungskonzept enthält dann verschiedene Sichten auf die Lösung (Kruchten 1995).

Die Erstellung *aspektbezogener Teilkonzepte* richtet sich nach den Regeln für den Entwurf des jeweiligen Aspekts, zum Beispiel den Regeln des Datenbankentwurfs für die Erstellung des Datenhaltungskonzepts. Auf eine nähere Ausführung wird hier verzichtet, da sie den Rahmen dieses Skripts sprengen würde.

Die Struktur des *Lösungskonzepts* (als Dokument) ist in Abschnitt 6.4. beschrieben. Wichtig ist, dass das Dokument nicht erst am Schluss der Entwurfstätigkeiten entsteht, sondern inkrementell während der gesamten Entwurfsarbeit.

6.5.3.3 Prüfung

Es ist zu *verifizieren*, dass das Konzept die in der Anforderungsspezifikation erfüllten Anforderungen erfüllt, dass es wirtschaftlich ist und softwaretechnisch gut. Kriterien für die softwaretechnische Güte sind zum Beispiel die Güte der Modularisierung (hohe Kohäsion, geringe Kopplung), die durchgängige Einhaltung des gewählten Entwurfsstils sowie Umfang und Verständlichkeit der Dokumentation.

Es ist sinnvoll, das Konzept zusätzlich auch zu *validieren*, d.h. unter Beteiligung der zukünftigen Benutzer festzustellen, ob es ihren Wünschen und Vorstellungen entspricht. Da das Konzept viel konkreter ist als die Anforderungsspezifikation, werden so möglicherweise noch Anforderungsfehler erkannt, bevor mit der Realisierung des Systems begonnen wird.

Das Mittel der Wahl zur Prüfung eines Lösungskonzepts ist das Review (vgl. Kapitel 9). Zur Prüfung kritischer Leistungsanforderungen sind evtl. Simulationen oder Labormuster (eine Prototypart, vgl. Kapitel 4) erforderlich. Auch Prototypen im engeren Sinn können zur Validierung herangezogen werden, zum Beispiel, um die Brauchbarkeit eines Benutzerschnittstellenkonzepts zu überprüfen.

Informelle Prüfungen finden kontinuierlich während der gesamten Konzipierungsarbeit statt. Vor Abschluss der Arbeit wird ein formelles Review durchgeführt und das Konzept entsprechend den erhobenen Befunden revidiert. Bei größeren Aufgaben werden Teillösungen schon vorab formellen Reviews unterzogen.

6.5.4 Variantenbehandlung

Es ist wesentlich, nicht stur einen einzigen Lösungspfad zu verfolgen, sondern stets den ganzen Raum möglicher Lösungen zu betrachten. Überall, wo sich mehrere Lösungsmöglichkeiten ergeben, werden die möglichen Lösungsvarianten soweit verfolgt, bis eine sinnvolle Entscheidung für die beste Variante möglich ist. Das Lösungskonzept, welches das Resultat der Konzipierung dokumentiert, enthält keine Varianten mehr. Es dokumentiert lediglich wichtige verworfene Varianten kurz, um die getroffenen Entscheidungen nachvollziehbar zu machen.

Bei den Kosten dürfen nicht nur die unmittelbaren Entwicklungskosten einer Variante betrachtet werden. Vielmehr müssen auch die Auswirkungen auf die Betriebs- und Pflegekosten sowie mögliche Folgekosten durch Einflüsse auf andere Komponenten untersucht werden.

Für die Untersuchung von Lösungsvarianten darf jedoch nicht beliebig viel Aufwand getrieben werden. Die Kosteneinsparungen durch die Wahl einer besseren bzw. der besten Variante müssen stets im richtigen Verhältnis zu den Kosten für die Untersuchung der Varianten stehen. Als Faustregel gilt: Je größer bzw. teurer der Untersuchungsgegenstand ist, desto aufwendiger darf die Untersuchung sein.

6.5.5 Vorgehen bei Beschaffung und Wiederverwendung

Die nachfolgend aufgeführten fünf Schritte mögen als grober Leitfaden für das Vorgehen bei *Beschaffungen* dienen. Der Beschaffungsprozess läuft parallel zur Spezifikation der Anforderungen und zur Konzipierung der Lösung. Der Aufwand, den man für die Beschaffung einer Komponente (bzw. eines Systems) treibt, hängt stark davon ab

- wie wichtig die Anwendung ist, für welche die Komponente beschafft wird
- wie teuer die beschaffte Komponente ist
- wie lange man die Komponente nutzen will.

Die Schritte 1 bis 3 sollten parallel zur Spezifikation der Anforderungen erfolgen. Die Schritte 4 und 5 (und evtl. ein Teil von 3) sind Teilaufgaben der Konzipierung.

1. **Anforderungen analysieren:** Wer seriös beschaffen will, muss als erstes die zu erfüllenden Anforderungen kennen.
2. **Hauptkriterien definieren:** Aus der Menge aller Anforderungen werden diejenigen herausgeschält, die kritisch und unverzichtbar sind (Faustregel: weniger als zehn, zum Beispiel drei bis fünf funktionale Anforderungen, zwei Leistungsanforderungen, der Maximalpreis und (bei reinen Software-Beschaffungen) die Hardware, auf der das System laufen muss).
3. **Marktübersicht verschaffen, Grobauswahl treffen:** Man muss herausfinden, was es für den gewünschten Problembereich überhaupt gibt. Quellen sind zum Beispiel Anzeigen,

Recherchen im WWW, Berichte und Tests in Zeitschriften, Händler sowie Kontakte zu anderen Unternehmen der gleichen Branche. Anhand der Kriterienliste wird eine Grobauswahl getroffen, bei der maximal noch drei bis fünf Kandidaten übrigbleiben sollten.

4. **Kandidatensysteme evaluieren:**

- Über die in die engere Wahl gekommenen Komponenten bzw. Systeme beschafft man sich genaue Unterlagen und lässt sie sich ggf. vorführen.
- Aussagen zu kritischen Punkten, die aus Werbematerial oder von Verkäufern stammen, sollten nachgeprüft werden.
- Bei größeren und teureren Komponenten bzw. Systemen ist eine formale Evaluation sinnvoll. Dabei werden alle Anforderungen gewichtet, für jede Anforderung und jedes Kandidatensystem die Erfüllung geprüft und die Ergebnisse zusammengezählt. Der Kandidat mit den meisten Punkten erfüllt den Anforderungskatalog am besten.
- Neben der formalen Bewertung sollten auch folgende Aspekte zur Beurteilung herangezogen werden
 - Vertrauenswürdigkeit des Herstellers/Händlers
 - Verfügbarkeit von Service
 - Erfahrungen mit anderen Produkten des gleichen Herstellers
 - Eigene Beschaffungspolitik (z.B. möglichst alles aus einer Hand beschaffen)
- Für jeden Kandidaten ist außerdem zu prüfen, welcher (finanzielle und zeitliche) Aufwand für notwendige Parametrierungen oder Anpassungsentwicklungen erforderlich ist.
- Sind Komponenten bzw. Systeme in einer Preisklasse über Fr. 10.000 zu beschaffen, empfiehlt sich die Miete einer Probeinstallation, welche eine intensive Erprobung unter realen Bedingungen ermöglicht. Dabei werden manchmal auch noch Probleme entdeckt, die bei der Formulierung der Anforderungen vergessen wurden.

5. **Entscheidung fällen und dokumentieren:** Aufgrund der Evaluation und evtl. Erprobung wird die Beschaffungsentscheidung gefällt. Die Gründe sollten kurz schriftlich festgehalten werden, damit die Entscheidung später noch nachvollziehbar ist.

Erfüllt keiner der evaluierten Kandidaten die gestellten Anforderungen, so ist als erstes zu prüfen, ob der Anforderungskatalog zu streng ist und wo man ggf. Abstriche machen kann. Führt dieser Weg nicht zum Ziel, so ist eine Beschaffung nicht möglich. In diesem Fall muss eine Lösung selbst entwickelt werden.

Bei *Wiederverwendung* werden in der Regel nur Komponenten, keine ganzen Systeme betrachtet. Das Vorgehen ist grundsätzlich gleich. Im Schritt 3 tritt an die Stelle der Marktübersicht die unternehmensinterne Suche nach wiederverwendbaren Komponenten. Es ist zu beachten, dass auch bei wiederverwendeter Software meistens Anpassungs- und Parametrierungsarbeiten anfallen, die etwas kosten. Ferner ist für jede wiederverwendete Komponente die Frage der Pflegeverantwortung (Lieferant oder Verwender) und der dabei möglicherweise entstehenden Kosten für Pflegeverträge zu klären.

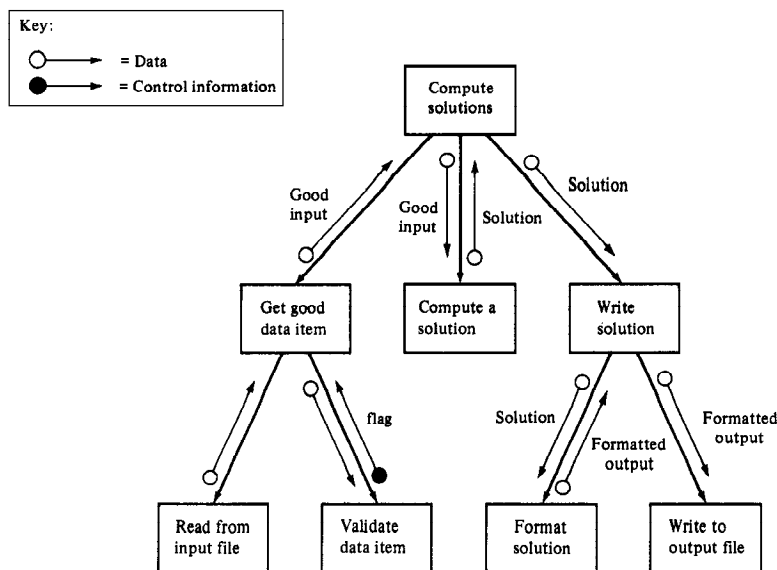
6.6 **Ausgewählte Architekturstile**

Eine Architektur ist im wesentlichen durch die Art der verwendeten Module, die Art(en) der Kooperation zwischen den Modulen und die Art der Modularisierung bestimmt. Sind diese drei Arten aufeinander abgestimmt, entsteht ein kohärenter *Architekturstil*. In den folgenden Abschnitten werden einige ausgewählte Architekturstile skizziert.

6.6.1 Funktionsorientierte Architektur (Structured Design)

Structured Design (Stevens, Myers, Constantine 1974; Page-Jones 1988) ist der klassische Vertreter einer *funktionsorientierten* Architektur. Die Grundidee ist, dass jeder Modul des Systems *eine Funktion* berechnet. Zur Realisierung einer Funktion können andere, einfachere Funktionen aufgerufen werden. Auf diese Art und Weise entsteht eine Hierarchie von Funktionen, wobei unten die elementaren und weiter oben die komplexeren Funktionen stehen. Die Funktion an der Spitze der Hierarchie ist das Hauptprogramm. Der Datenaustausch zwischen den Modulen erfolgt entweder bei Aufruf und Rückkehr durch Parameter oder durch Zugriff auf explizit modellierte gemeinsame Speicherbereiche. Bei den Parametern wird teilweise noch zwischen echten Daten und Steuerinformation unterschieden. Häufig werden die Funktionen zusätzlich in Eingabe-, Verarbeitungs- und Ausgabefunktionen gegliedert und die Hierarchie entsprechend strukturiert (Bild 6.2).

Eine funktionsorientierte Modularisierung kann den in Abschnitt 6.3.2 genannten Kriterien nur teilweise genügen. Alle Aufgaben, bei denen eine Gruppe von Funktionen einen gemeinsamen Status verwaltet (zum Beispiel Treiber, Datenstrukturen oder intern kooperierende Anwendungsfunktionen), lassen sich nicht geeignet behandeln. Solche Funktionsgruppen müssten nämlich nach den Regeln von Abschnitt 6.3.2 zusammen mit den Statusdaten zu einem Modul zusammengefasst werden, was jedoch mit Modulen, die nur *eine* Funktion enthalten (und damit nur einen möglichen Aufruf haben) nicht möglich ist. Diese ungenügenden Abstraktionsmöglichkeiten erschweren das Verständnis und die Änderbarkeit großer funktionsorientierter Entwürfe massiv.



Quelle: Fairley (1985)

Bild 6.2. Funktionsorientiert gegliedertes System

6.6.2 Datenorientierte Architektur (Entwurf mit Datenabstraktionen)

Im Zentrum der datenorientierten Architektur steht eine Modularisierung nach dem Geheimnisprinzip (vgl. 6.3.3). Wie bereits dort erwähnt, gibt es dabei vier typische Arten von Entwurfsentscheidungen, die zu Modulen führen:

- wie eine Funktion realisiert ist,
- wie ein Objekt aus dem Anwendungsbereich repräsentiert und realisiert ist,
- wie eine Datenstruktur aufgebaut ist und wie sie bearbeitet werden kann,
- wie Leistungen Dritter realisiert sind.

Die Fälle (b), (c) und (d) benötigen einen Abstraktionsmechanismus, welcher eine Datenstruktur und die auf den Daten dieser Struktur möglichen Operationen zu einer Einheit zusammenfasst. Eine solche Abstraktion wird *Datenabstraktion* genannt. *Abstrakte Datentypen* sind das bekannteste Mittel zum Aufbau von Datenabstraktionen. Der im Kapitel 7.5.4 spezifizierte Stack ist ein typisches Beispiel für einen abstrakten Datentyp.

Bei der datenorientierten Architektur besteht ein System aus einer Menge von Datenabstraktionen, die in ähnlicher Weise wie bei einer funktionsorientierten Architektur aufeinander aufbauen. Es entsteht so eine *Benutzungshierarchie*: Die Datenabstraktionen benutzen in ihren Implementierungen Operationen, die von tieferliegenden Datenabstraktionen angeboten werden. In der Regel versucht man, die Datenabstraktionen in Schichten zu gliedern (Anwendung der Virtuelle Maschinen-Metapher, vgl. 6.8.2) Bild 6.3 zeigt als Beispiel die Architektur der Software für einen Bancomat. Man beachte, dass ein Pfeil in diesem Bild nicht wie in Bild 6.2 den Aufruf einer einzelnen Funktion, sondern die Benutzung von Leistungen bezeichnet. Die Zusammenarbeit zwischen Leistungsanbietern und Leistungsverwendern kann durch Verträge spezifiziert werden (Design by Contract).

Nach diesem Stil entworfene Systeme sind aufgrund der Eigenschaften des Geheimnisprinzips leicht änderbar und vor allem im Großen gut verstehbar. Ein datenorientierter Entwurf führt damit genau auf Modularisierungen, wie sie gemäß Abschnitt 6.3.2 erwünscht sind.

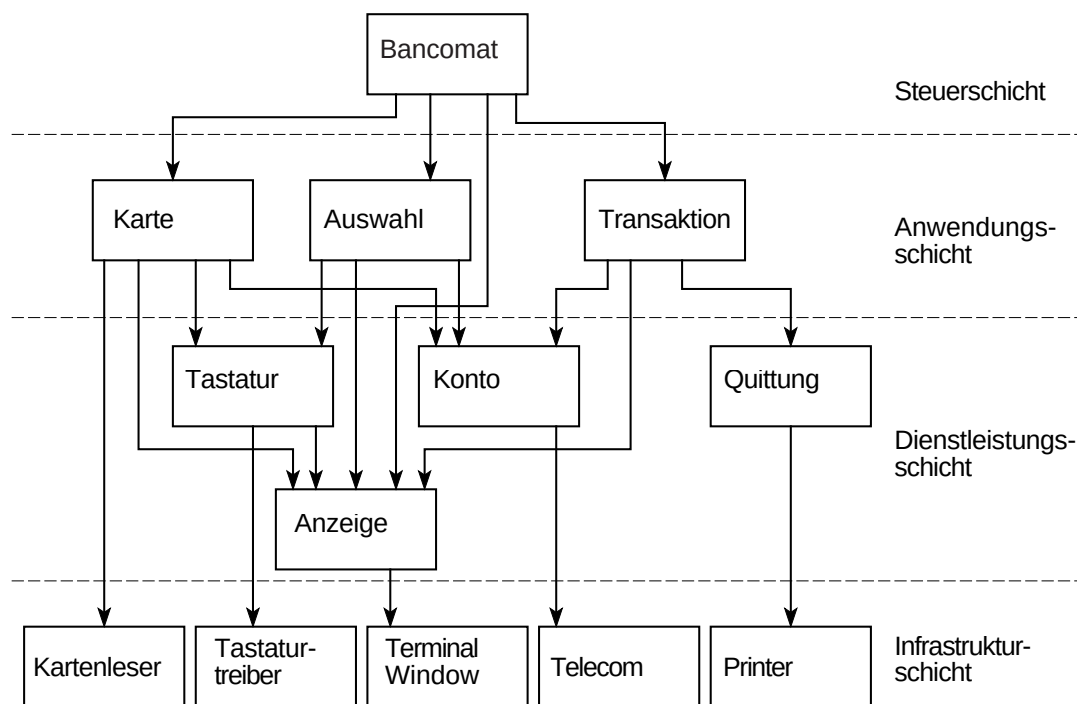


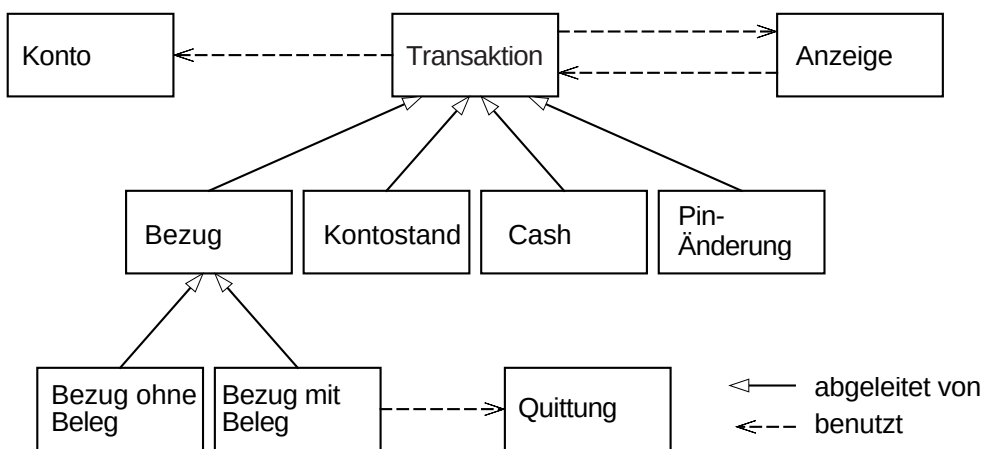
Bild 6.3. Benutzungshierarchie und Schichtung im Entwurf mit Datenabstraktionen

6.6.3 Objektorientierte Architektur

Im objektorientierten Entwurf wird ein System als eine Menge kooperierender Objekte aufgefasst. Wie eine Datenabstraktion besteht ein Objekt aus Daten und allen darauf möglichen Operationen. Objekte bzw. Klassen (d.h. Objekttypen) bilden die Einheiten der Modularisierung. Jedes Objekt repräsentiert ein Objekt aus dem Anwendungsbereich (zum Beispiel einen Kunden oder einen Artikel) oder ein in sich geschlossenes Informatik-Element, das für die Problemlösung benötigt wird (zum Beispiel ein Dialogfenster oder einen Schaltknopf in der Benutzerschnittstelle). Das Verbergen des Objektinneren nach dem Geheimnisprinzip ist teilweise

möglich und wird angestrebt. Das Prinzip der Vererbung (s.u.) lässt jedoch ein konsequentes Information Hiding nicht zu.

Die Objekte können auf zwei verschiedene Arten kooperieren. Erstens durch *Benutzung*: Ein Objekt benutzt in seiner Implementierung Operationen, die von anderen Objekten angeboten werden. Im Gegensatz zum Entwurf mit Datenabstraktionen wird keine Benutzungshierarchie angestrebt. Zweitens durch *Vererbung*: Von jeder Klasse können *Unterklassen* abgeleitet werden, welche alle Eigenschaften (Datenstrukturen und Operationen) von der Originalklasse übernehmen. Man sagt auch, eine Klasse *vererbt* ihre Merkmale an ihre Unterklassen. Die Möglichkeit der Vererbung macht die Entwürfe außerordentlich flexibel und eröffnet neue Möglichkeiten der Wiederverwendung. Der Entwurf und die Realisierung einer Unterklasse erfordern in der Regel jedoch eine zumindest teilweise Kenntnis der inneren Struktur der Klasse, von der abgeleitet wird. Information Hiding ist daher zwischen Ober- und Unterklassen nur begrenzt möglich.



Erläuterung: Modelliert ist ein Ausschnitt aus einem Bancomat-System. Die möglichen Transaktionen sind als Unterklassen von TRANSAKTION modelliert. Dies erlaubt die flexible Erweiterung der Software um weitere Transaktionsarten (zum Beispiel Überweisung tätigen). TRANSAKTION benutzt KONTO für Kontostandsabfragen und -mutationen. BEZUG MIT BELEG benutzt QUITTUNG zum Erstellen von Belegen. Die Interaktion zwischen TRANSAKTION und ANZEIGE ist nach dem Beobachtermuster (vgl. Bild 6.7) konstruiert: TRANSAKTION benachrichtigt ANZEIGE jedesmal, wenn innerhalb von TRANSAKTION oder von einer ihrer Unterklassen eine Veränderung stattgefunden hat, welche die Anzeige beeinflusst. ANZEIGE holt sich daraufhin die benötigte Information ab und zeigt sie an.

Bild 6.4. Benutzung und Vererbung in einer objektorientierten Architektur

Durch die Verwendung von zwei verschiedenen Kooperationsmechanismen (Benutzung und Vererbung) ist der objektorientierte Entwurf methodisch anspruchsvoller als traditionelle Entwurfsverfahren. Bei richtiger Anwendung ist das resultierende Lösungsmodell von hoher Qualität, indem es einerseits ein geradliniges Abbild des Modells der Aufgabenstellung darstellt und andererseits änderbar und erweiterbar ist.

Unsachgemäßer und undisziplinierter Gebrauch der Objektorientierung kann jedoch auch zu ausgesprochen schlechten Entwürfen führen. Labyrinthische Benutzungsgeflechte quer über alle Abstraktionsebenen und die Verwendung der Vererbung als Steinbruch (irgendwie Passendes wird durch Ableitung von Unterklassen übernommen, Unpassendes wird entfernt oder undefiniert) führen zu nebenwirkungsreichen, schwer verstehbaren Systemen, deren Pflege extrem aufwendig werden kann.

6.6.4 Prozessorientierte Architektur

Nebenläufige Systeme bestehen aus einer Menge unabhängig arbeitender, untereinander kooperierender Akteure, die typisch als Prozesse realisiert sind. Die Prozesse kooperieren durch

Austausch von Nachrichten oder durch Zugriff auf gemeinsame Speicherbereiche. Die Prozesse können auf einem Rechner ablaufen oder auf verschiedene Rechner verteilt sein.

In prozessorientierten Architekturen sind die Prozesse die Module der obersten Stufe. Jeder Prozess ist typisch ein *sequentiell* ablaufender Systemteil, der seinerseits modularisiert ist, beispielsweise in objektorientiertem Stil.

Neben der Prozessstruktur und den Architekturen der einzelnen Prozesse muss auch eine Kommunikationsarchitektur entworfen werden. Heute erleichtert man sich diese Aufgabe in der Regel durch die Verwendung käuflicher Kommunikationssoftware, welche den Prozessen komfortable Kommunikationsschnittstellen zur Verfügung stellt. Die Prozesskommunikation wird so von einer Dienstleistungsschicht übernommen, deren Architektur und Realisierung vor den Verwendern verborgen bleibt.

Bild 6.5 zeigt als Beispiel einer prozessorientierten Architektur die Erfassung, Bereitstellung und Archivierung von Wechselkursdaten in einer Bank oder einem Handelshaus.

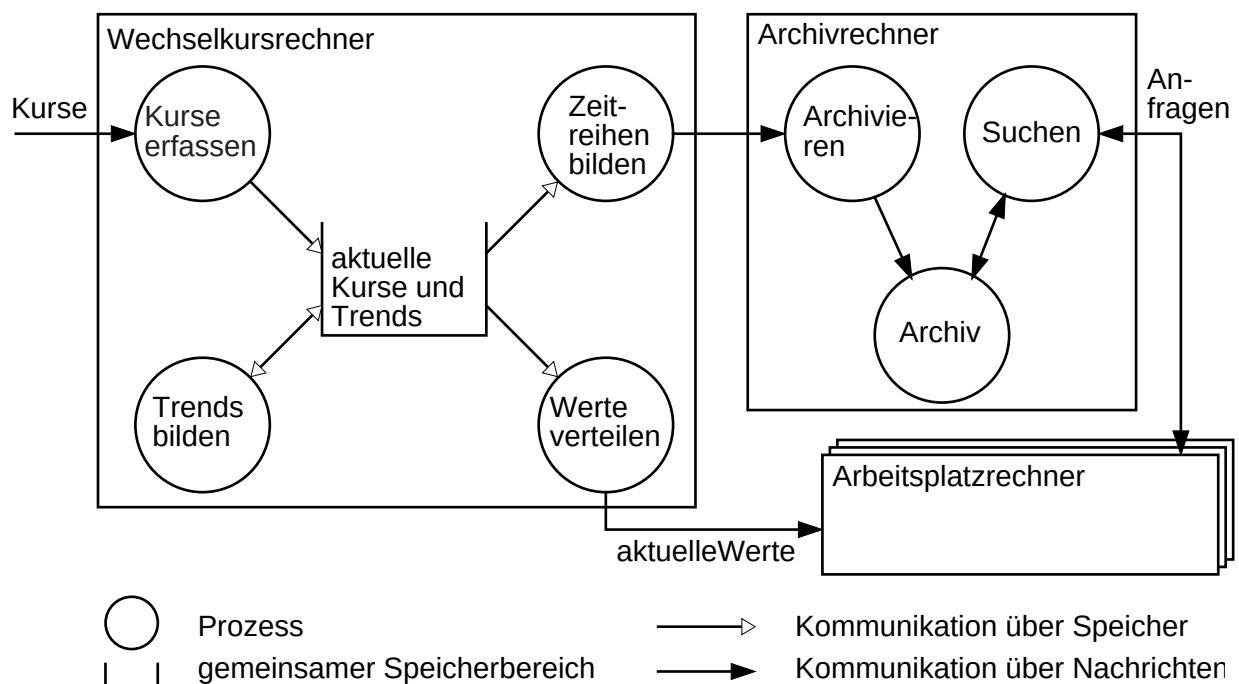


Bild 6.5. Beispiel einer prozessorientierten Architektur

6.6.5 Komponentenorientierte Architektur

Die komponentenorientierte Architektur ist eine Weiterentwicklung der objektorientierten Architektur, insbesondere im Hinblick auf Verteilung und Wiederverwendung. Es zeigt sich nämlich, dass Klassen als Einheiten für die geographische Verteilung eines Systems und als Einheiten käuflicher, wiederverwendbarer Software oft zu klein sind.

In einer komponentenorientierten Architektur versteht man unter einer *Komponente* eine Menge zusammengehöriger Objekte bzw. zusammengehöriger Klassen, die von ihrer Umgebung abgekapselt und nur über eine oder mehrere Schnittstelle(n) der Komponente zugänglich sind. Komponenten in diesem Sinn sind nach dem Geheimnisprinzip gebildete Module auf einer Stufe oberhalb von Objekten und Klassen, die zudem geographisch verteilt sein können.

Die Komponenten kommunizieren miteinander über einen *Makler* (broker): Einerseits geben sie nach außen eine Schnittstelle bekannt, über die sie angesprochen werden können. Andererseits greifen sie mit Hilfe des Maklers auf Objekte anderer Komponenten zu, indem sie in deren Schnittstelle definierte Operationen aufrufen (Bild 6.6). Die Komponenten sind stark voneinan-

der entkoppelt. Sie brauchen weder die Art der Realisierung der Kommunikation durch den Makler, noch die geographische Lokalisierung der Partnerkomponenten noch deren Implementierung zu kennen. Wenn die Schnittstellen in einer eigenen Schnittstellenbeschreibungssprache (interface definition language, IDL) beschrieben sind, können die Komponenten sogar in unterschiedlichen Programmiersprachen realisiert sein.

Ein typischer Vertreter einer komponentenorientierten Architektur ist die sogenannte *Client/Server-Architektur*. Eine Client/Server-Architektur besteht aus einer Menge von Klienten (Clients), d.h. Anwendungsprogrammen, welche auf Arbeitsplatzstationen laufen, einer Menge von Lieferanten (Servers), die Daten bereitstellen (Datenbanksysteme, Dateisysteme) und einer Kommunikationsschiene, welche alle Komponenten miteinander verbindet. Erweitert man die Kommunikationsschiene zu einer Zwischenschicht, welche Kommunikations-, Koordinations- und Übersetzungsdienste sowie ausgewählte Teile der Anwendung realisiert, so spricht man von einer *Middleware-Architektur*. (Tresch 1996).

Komponentenorientierte Architekturen werden manchmal auch als *Architektur mit verteilten Objekten* bezeichnet.

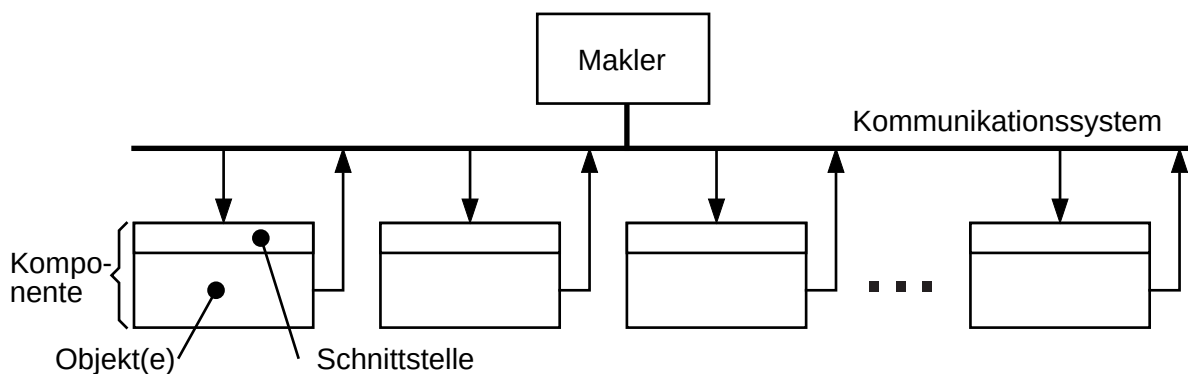


Bild 6.6. Beispiel einer komponentenorientierten Architektur

6.7 Wiederverwendung von Architekturen und Entwurfswissen

6.7.1 Entwurfsmuster

Entwurfsmuster (Gamma et al. 1995) stellen bewährte, vorgefertigte Lösungsschablonen für wiederkehrende Entwurfsprobleme bereit und ermöglichen damit die Wiederverwendung von Entwurfswissen. Sie schaffen ferner eine begriffliche Basis und Terminologie für die Kommunikation über solche Probleme.

DEFINITION 6.10. *Entwurfsmuster (design pattern).* Eine spezielle Komponente, die eine allgemeine, parametrierbare Lösung für ein typisches Entwurfsproblem bereitstellt.

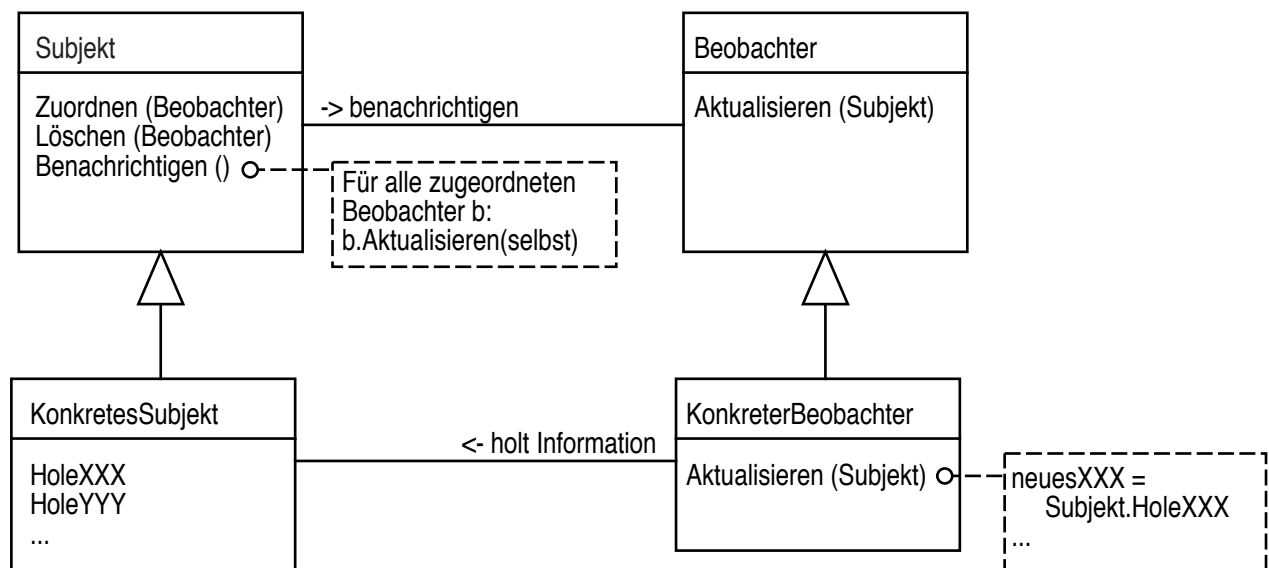
Grundsätzlich gibt es eine Fülle von Mustern, sowohl allgemeiner Art als auch für spezifische Anwendungsbereiche. *Musterkataloge* erschließen das Wissen über Muster und machen die Muster wiederverwendbar.

Entwerfen mit Mustern setzt voraus, dass die Entwerfenden einen Grundschatz an Mustern kennen. Bei der Modularisierung müssen Anwendungssituationen für Muster erkannt und die entsprechenden Muster verwendet werden.

Bild 6.9 zeigt das *Beobachtermuster* (observer pattern, Gamma et al. 1995) als Beispiel. Die Intention des Beobachtermusters ist, dass Objekte Daten bei einem Informationsanbieter (Sub-

jekt) abonnieren können. Bei jeder Änderung der abonnierten Daten werden die Abonnenten (Beobachter) automatisch über die Änderung informiert bzw. werden ihnen die geänderten Daten zugestellt. Dieses Muster hat zwei typische Anwendungen:

- Entkopplung voneinander abhängiger Objekte durch Ersetzung direkter Kommunikation über Aufrufe durch eine indirekte über Benachrichtigung
- Trennung von Informationsbereitsteller und einer Menge von Informationsverarbeitern bzw. -darstellern.



- 1 ZUORDNEN fügt ein Beobachterobjekt in die Liste der zu benachrichtigenden Beobachter ein.
- 2 Nach jeder Veränderung ruft ein KONKRETESSUBJEKT seine (geerbte) Operation BENACHRICHTIGEN auf. Diese iteriert die Liste der Beobachter und schickt jedem Beobachter die Botschaft AKTUALISIEREN. Das benachrichtigende Subjekt wird als Parameter mitgegeben.
- 3 Jeder benachrichtigte Beobachter reagiert, indem er beim benachrichtigenden Subjekt mit den Botschaften HOLEXXX, HOLEYYY etc. die ihn interessierenden Informationen abrufen.

Varianten:

- Der Botschaft AKTUALISIEREN können weitere Informationen (z.B. die Art des eingetretenen Ereignisses) als Parameter mitgegeben werden.
- Der Botschaft AKTUALISIEREN können die veränderten Daten gleich mitgegeben werden, wodurch der Abruf durch HOLEXXX entfällt. Dieses Bringprinzip ist effizient, koppelt aber Gegenstand und Beobachter stärker als das mit HOLEXXX realisierte Holprinzip.

Bild 6.7. Das Beobachtermuster

6.7.2 Rahmen

Rahmen (frameworks) stellen vorgefertigte Lösungsgerüste für bestimmte Problemklassen bereit und gestatten damit die Wiederverwendung ganzer Architekturen.

DEFINITION 6.11. *Rahmen (framework).* Eine Menge kooperierender Module, die das Grundgerüst für die Lösung einer bestimmten Klasse von Problemen bilden.

Konkrete Lösungen entstehen durch Ergänzung oder Spezialisierung des Rahmens durch problemspezifische Module. Der Umfang eines Rahmens kann stark variieren:

- Enge Rahmen lösen ein spezifisches Teilproblem, zum Beispiel die Realisierung einer Benutzerschnittstelle, die Verwaltung von Dateien mit Vergabe und Prüfung von Zugriffsrechten oder die Grundfunktionalität eines Editors.

- Umfassende Rahmen bilden das Lösungsgerüst für ein vollständiges System, zum Beispiel ein Rahmen für das Schaltergeschäft in einer Bank.

Manchmal werden in einer Architektur mehrere Rahmen gleichzeitig verwendet. Dabei muss der Vorteil der Wiederverwendung allerdings sorgfältig abgewogen werden gegen die Nachteile möglicher Architekturkonflikte zwischen den verschiedenen Rahmen. Die Einbettung unterschiedlicher Architekturprinzipien der verschiedenen Rahmen in eine gemeinsame, kohärente Gesamtarchitektur kann sehr schwierig bis sogar unmöglich sein.

6.7.3 Architekturmuster

Analog zu Entwurfsmustern sind Architekturmuster vorgefertigte Strukturen für typische Architekturprobleme.

Definition 6.12. *Architekturmuster.* Eine vorgefertigte, parametrierbare Schablone für die Gestaltung der Architektur eines Systems oder einer Komponente.

Es gibt eine Fülle von Architekturmustern. Man kann unterscheiden zwischen Strukturmustern, Steuermustern und Modularisierungs-/Entkopplungsmustern. Nachfolgend werden einige ausgewählte Architekturmuster beschrieben.

6.7.3.1 Strukturmuster: Das Matrixmuster

Das Matrixmuster (Bild 6.8) gliedert ein System in je eine Menge von Datenmodulen und von Funktionsmodulen, wobei jede Funktion auf jedes Datum zugreifen kann. Die Funktionsmodule enthalten keine permanenten Daten. Typische Anwendungen dieses Musters findet man in der klassischen Architektur datenbankbasierter Systeme.

6.7.3.2 Steuermuster

Das EVA (Eingabe-Verarbeitung-Ausgabe)-Muster

Ein Steuermodul steuert nacheinander (in Sequenz oder iterativ) Eingabe-, Verarbeitungs- und Ausgabemodule an. Dies ist das klassische Steuermuster für funktionsorientierte sequentielle Programme (vgl. Bild 6.2).

Das Hauptschleifenmuster

Ein Prozess misst, regelt und steuert, indem er in einer Endlosschleife zyklisch alle Datenquellen (Sensoren, etc.) abfragt und alle Datensinken (Anzeigen, Aktuatoren...) mit aktualisierten Werten versorgt. Dieses Muster wird wegen seiner Robustheit gerne für die Steuerung sicherheitskritischer Anwendungen verwendet.

Das Hollywood-Muster (“Don’t call us, we call you”)

Ein Ereignisverwalter registriert alle Eingabeereignisse und ruft die zugehörigen Dienste auf. Das Anwendungsprogramm enthält kein Hauptprogramm mehr. Dieses Muster liegt den meisten käuflichen Anwendungsrahmen zugrunde.

6.7.3.3 Modularisierungs-/Entkopplungsmuster: Das Model-View-Controller (MVC)-Muster

Ein System wird in drei Teile gegliedert: Das Modell (“Model”) enthält die Anwendungslogik und das Modell des Anwendungsbereichs, die Sicht (“View”) realisiert die äußere, sichtbare Repräsentation des Systems und die Steuerung (“Controller”) behandelt alle Benutzereinga-

ben (Krasner und Pope, 1988). Mit diesem Muster werden die Anwendungslogik, die Repräsentation der Anwendung gegenüber der Außenwelt und die Steuerung der Eingaben voneinander entkoppelt. Das MVC-Muster ist ein zentrales Muster im Entwurf objektorientierter Systeme. Es kann auch im Kleinen als Entwurfsmuster verwendet werden (Gamma et al. 1995).

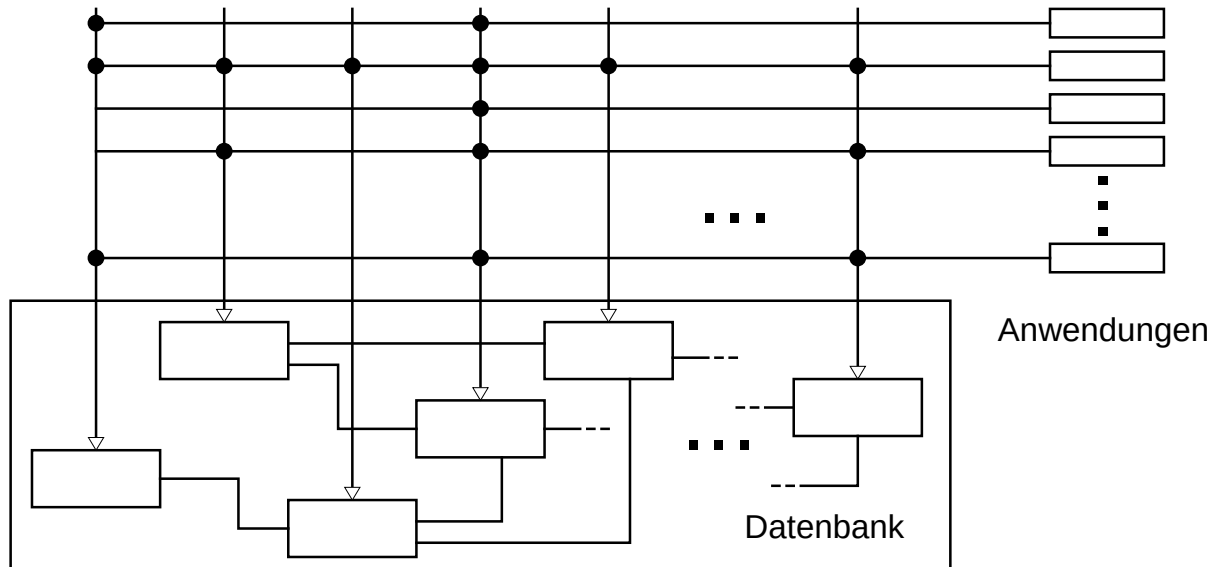


Bild 6.8. Architektur eines datenbankbasierten Anwendungssystems nach dem Matrixmuster

6.8 Architekturmetaphern

Eine *Metapher* ist ein sprachlicher Ausdruck, bei dem ein Wort aus seinem Bedeutungszusammenhang in einen anderen übertragen, *als Bild verwendet wird*.

Definition 6.13. *Architekturmetapher.* Leitbild für die Gestaltung einer Architektur.

Eine *Architekturmetapher* ist ein Modell, das eine Architektur über analoge, vertraute Bilder erschließt. Die Metapher ermöglicht ein besseres Verständnis der Systemstruktur und ist das Leitbild für die Gestaltung der Architektur. In den folgenden Abschnitten werden einige Architekturmetaphern kurz vorgestellt.

6.8.1 Die WAM (Werkzeug-Automat-Material)-Metapher

Das System besteht aus *Materialien* (fachliche Arbeitsgegenstände der Anwendung), die von den Benutzern mit passenden *Werkzeugen* interaktiv bearbeitet werden. Vollständig automatisierbare Routineaufgaben werden von *Automaten* erledigt (Züllighoven, 1998).

- Werkzeuge sind gegenüber den Materialien aktiv, indem sie Materialien bearbeiten. Sie werden von Menschen benutzt bzw. bedient. Diesen gegenüber verhalten sie sich passiv oder assistierend; sie überlassen die Kontrolle immer den bedienenden Menschen.
- Materialien sind passiv und/oder speichernd. Sie werden bearbeitet und können sowohl Arbeitsgegenstand als auch Arbeitsergebnis sein.
- Automaten sind aktiv. Sie erledigen Aufgaben vollautomatisch und ohne menschliches Zutun.

Die WAM-Metapher eignet sich insbesondere für die Architektur von Assistenzsystemen, d.h. Systemen, welche kreative menschliche Arbeit (zum Beispiel Beratungs- oder Entwurfstätigkeiten) unterstützen.

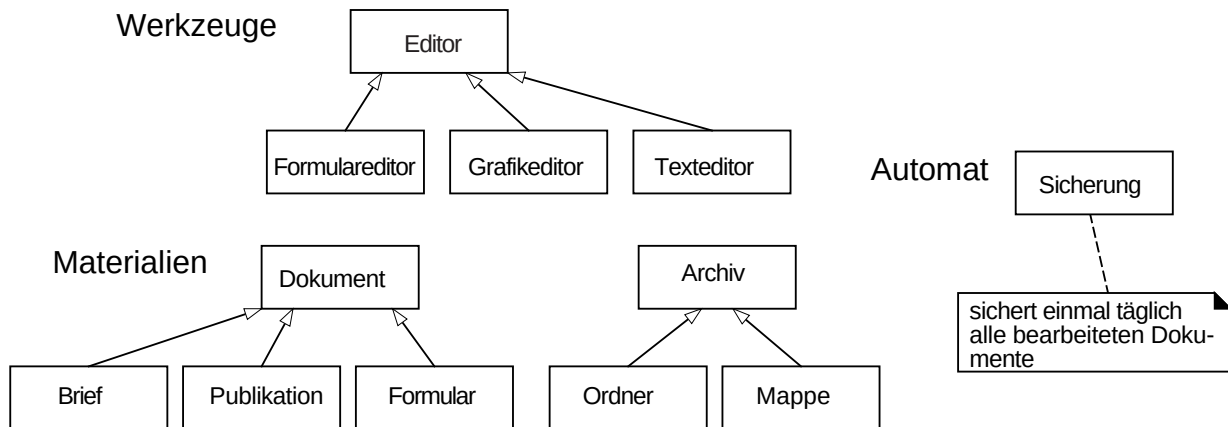


Bild 6.9. Die WAM (Werkzeug-Automat-Material)-Metapher

6.8.2 Weitere Architekturmetaphern

- **Die Organisationshierarchie.** Die Architektur wird analog zur Organisationshierarchie in einem Unternehmen strukturiert, mit Delegation von Aufgaben und Verantwortung von oben nach unten und Berichten von unten nach oben.
- **Die virtuellen Maschinen.** Die Architektur ist als eine Menge aufeinander aufbauender Schichten von *virtuellen*, d.h. künstlichen Maschinen aufgebaut. Die Maschinen jeder Schicht bieten Leistungen für die darüberliegende Schicht an und benutzen dazu die Maschinen der darunterliegenden Schicht. Kommunikationsarchitekturen sind meistens nach dieser Metapher konstruiert.
- **Das Steckersystem.** Die Architektur wird in Analogie zu technischen Steckersystemen aufgebaut. In einen Grundrahmen, welcher typisch Kommunikations-, Dialog, und Datenverwaltungsdienste anbietet, werden Anwendungselemente „eingesteckt“. Anwendungsrahmen (vgl. 6.7.2) sind oft nach dieser Metapher konstruiert (Bild 6.10).

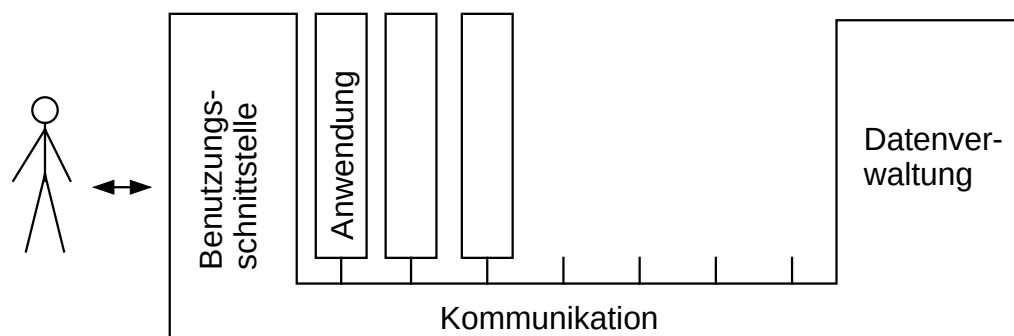


Bild 6.10. Die Steckersystem-Metapher

Aufgaben

- 6.1** Welche Vorteile hat eine Modularisierung nach dem Geheimnisprinzip (a) für die Verwen-
der eines Moduls, (b) für die Modulersteller/für das Pflegepersonal?
- 6.2** Gegeben sei folgende Modularisierung der Steuerung einer Abfüllanlage für Flüssigkeiten:
Modul Tank: Steuerung des Einlassventils, Behandlung der Sensoren für vollen und leeren
Tank, Meldung des aktuellen Tankzustands an Leitstand
Modul Abfüllventil: Steuerung des Auslassventils am Tank
Modul: Abfüllung: Steuerung des Fließbands mit den abzufüllenden Behältern, Behand-
lung des Waage-Sensors (misst die abgefüllte Menge)
Randbedingungen: 1. Bei leerem Tank darf das Auslassventil nicht geöffnet sein. 2. Wäh-
rend der Nachfüllung des Tanks (Einlassventil offen) muss das Auslassventil geschlossen
sein.
Beurteilen Sie diese Modularisierung, insbesondere im Hinblick auf Information Hiding
und Zusammenarbeit.

Ergänzende und vertiefende Literatur

Fairley (1985) sowie Ghezzi, Jazayeri und Mandrioli (1991) beschreiben die Prinzipien guten Entwurfs.

McDermid (1991) enthält eine Übersicht über klassische (nicht-objektorientierte) Entwurfsver-
fahren.

Das Tutorium von Freeman und Wasserman (1983) gibt anhand von Originalartikeln einen
Überblick über verschiedene klassische Entwurfstechniken. Unter anderem ist auch Parnas
(1972) abgedruckt.

Meyer (1997) bzw. (1988) ist *das* klassische Lehrbuch über objektorientierten Entwurf. Weitere
lesenswerte Bücher über objektorientierten Entwurf sind Booch (1994), Wirfs-Brock, Wilkerson,
Wiener (1990) und Züllighoven (1998).

Shaw und Garlan (1996) geben einen Überblick über Software-Architektur und gehen insbeson-
dere auf verschiedene Architekturstile und auf Architekturbeschreibungssprachen ein. Garlan
und Shaw (1993) ist eine Kurzfassung, die sich hauptsächlich mit Architekturstilen beschäftigt.

Gamma, Helm, Johnson und Vlissides (1995) führen in das Konzept der Entwurfsmuster ein und
beschreiben 23 häufige und typische Muster im Detail.

Dijkstra (1968) beschreibt als erster eine Architektur, die aus mehreren aufeinander aufbauenden
Schichten besteht.

Parnas (1972) beschreibt als erster das Geheimnisprinzip (Information Hiding).

Züllighoven (1998) beschreibt ein in langjähriger Praxis erprobtes Vorgehen für die Konstruk-
tion objektorientierter Systeme.

Zitierte Literatur

Booch, G. (1994). *Object Oriented Analysis and Design with Applications*. Second Edition.
Redwood City, Ca.: Benjamin/Cummings.

Dijkstra E.W. (1968). The Structure of the THE multiprogramming System. *Communications of
the ACM* **11**, 5 (May 1968). 341-346.

Fairley, R.E. (1985). *Software Engineering Concepts*. New York, etc.: McGraw-Hill.

- Freeman, P., A. I. Wasserman (eds.) (1983). *Tutorial on Software Design Techniques*. IEEE Computer Society Press, Order Number 514.
- Gamma, E., R. Helm, R. Johnson, J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass. etc.: Assison-Wesley.
- Garlan, D. M. Shaw (1993). An Introduction to Software Architecture. In V. Ambriola and G. Tortora (eds.) *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, Singapore. 1-39 [Auch erschienen als technische Bericht von CMU bzw. SEI: CMU-CS-94-166, CMU/SEI-94-TR-21]
- Ghezzi, C., M. Jazayeri, D. Mandrioli (1991). *Fundamentals of Software Engineering*. Englewood Cliffs: Prentice-Hall.
- IEEE (1990). *Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990. IEEE Computer Society Press.
- Krasner, G.E., S.T. Pope (1988). A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming* **1**, 3. 26-49.
- Kruchten, P. (1995). The 4+1 View Model of Architecture. *IEEE Software* **12**, 6 (Nov 1995). 42-50.
- McDermid, J. (1991). *Software Engineer's Reference Book*. Oxford: Butterworth-Heinemann. ca. 1200 p.
- Meyer, B. (1988). *Object-Oriented Software Construction*. Englewood Cliffs, N.J.: Prentice Hall.
[auf Deutsch: *Objektorientierte Softwareentwicklung*, München: Hanser, 1990]
- Meyer, B. (1992). Applying "Design by Contract". *IEEE Computer* **25**, 10 (Oct. 1992). 40-51.
- Meyer, B. (1997). *Object-Oriented Software Construction*. Englewood Cliffs, N.J.: Prentice Hall. [Neuaufgabe von Meyer (1988)]
- Page-Jones, M. (1988). *The Practical Guide to Structured Systems Design*. Englewood Cliffs, N.J.: Prentice Hall.
- Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM* **15**, 12 (Dec. 1972). 1053-1058.
- Shaw, M., D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs, N.J.: Prentice Hall. 242 p.
- Stevens, W.G., G. Myers, L. Constantine (1974). Structured Design. *IBM Systems Journal* **13**, 2. 115-139.
- Tresch, M. (1996). Middleware: Schlüsseltechnologie zur Entwicklung verteilter Informationssysteme. *Informatik-Spektrum* **19**, 5 (Okt 1996). 249-256.
- Wirfs-Brock, R., B. Wilkerson, L. Wiener (1990). *Designing Object-Oriented Software*. Englewood Cliffs, N.J.: Prentice Hall.
[auf Deutsch: *Objektorientiertes Software Design*, München: Hanser, 1993]
- Züllighoven, H. (1998). *Das objektorientierte Konstruktionshandbuch*. Heidelberg: dpunkt Verlag.

7. Spezifikation von Anforderungen

7.1 Grundlagen und Motivation

7.1.1 Definitionen und grundlegende Begriffe

DEFINITION 7.1. *Anforderung (requirement).* (1) Eine Bedingung oder Fähigkeit, die von einer Person zur Lösung eines Problems oder zur Erreichung eines Ziels benötigt wird. (2) Eine Bedingung oder Fähigkeit, die eine Software erfüllen oder besitzen muß, um einen Vertrag, eine Norm oder ein anderes, formell bestimmtes Dokument zu erfüllen. (IEEE 610.12-1990)

DEFINITION 7.2. *Anforderungsspezifikation.* Die Zusammenstellung aller Anforderungen an eine Software. Synonyme: Anforderungsdokument, Software Requirements Specification.

Im Alltag ist die Sprechweise nicht immer ganz eindeutig: mit „die Spezifikation“ ist häufig je nach Kontext das resultierende *Dokument* oder der *Spezifikationsprozess* (das heißt der Prozess des Erfassens, Beschreibens und Prüfens von Anforderungen) gemeint.

Werden die Aufgaben im Bereich der Spezifikation und des Spezifizierens systematisch und gezielt angegangen, spricht man auch von *Requirements Engineering* (Anforderungstechnik). In Definition 7.3 geben wir zwei mögliche Definitionen, eine mehr technische, welche sich an die Definition von Software Engineering in IEEE 610.12-1990 anlehnt und eine mehr menschenorientierte, ähnlich derjenigen von Gause und Weinberg (1989).

DEFINITION 7.3. *Requirements Engineering (Anforderungstechnik).* 1. Das *systematische, disziplinierte* und *quantitativ erfassbare* Vorgehen beim Spezifizieren, d.h. Erfassen, Beschreiben und Prüfen von Anforderungen an Software. 2. *Verstehen* und *Beschreiben*, was die Kunden *wünschen* oder *brauchen*.

Im deutschen Sprachraum ist im Zusammenhang mit Anforderungen auch der Name *Pflichtenheft* gebräuchlich. Mit diesem Namen werden jedoch häufig verschiedene Begriffe verbunden. Manche Leute verstehen Pflichtenheft als ein Synonym zu Anforderungsspezifikation, andere verlangen eine grobe Beschreibung der gewählten Lösung als Bestandteil, wiederum andere sehen auch Elemente der Projektabwicklung (Kosten, Termine) als Bestandteil des Pflichtenhefts. Bei der Verwendung des Namens „Pflichtenheft“ ist daher Vorsicht geboten und das jeweils damit Gemeinte im Einzelfall zu klären.

7.1.2 Wozu eine Anforderungsspezifikation erstellen?

Die Erstellung einer Anforderungsspezifikation kostet Geld, ohne dass diesem Aufwand ein unmittelbar sichtbarer Ertrag in Form von Programmen gegenübersteht. Das Spezifizieren von Anforderungen ist also nur dann wirtschaftlich, wenn dem dafür zu treibenden Aufwand entsprechende Einsparungen gegenüberstehen.

Requirements Engineering, das systematische Spezifizieren von Anforderungen, hat daher das klare Ziel, *Kosten zu senken*.

Dass dieses Ziel realistisch ist, zeigt folgende Überlegung: *Fehlerkosten*, d.h. die Kosten für die Lokalisierung und Behebung von Fehlern, machen einen wesentlichen Teil der Gesamtkosten einer Systementwicklung aus. Anforderungsfehler sind dabei die teuersten Fehler,

weil sie beim Fehlen einer Anforderungsspezifikation typisch erst bei der Abnahme oder im Betrieb gefunden werden und die Fehlerkosten exponentiell mit der Verweildauer der Fehler im System wachsen (Bild 7.1). Wenn man Requirements Engineering vernünftig betreibt, sind die Einsparungen bei den Fehlerkosten höher als der dafür notwendige Aufwand. Das heißt, das Spezifizieren von Anforderungen ist *wirtschaftlich* (Bild 7.2).

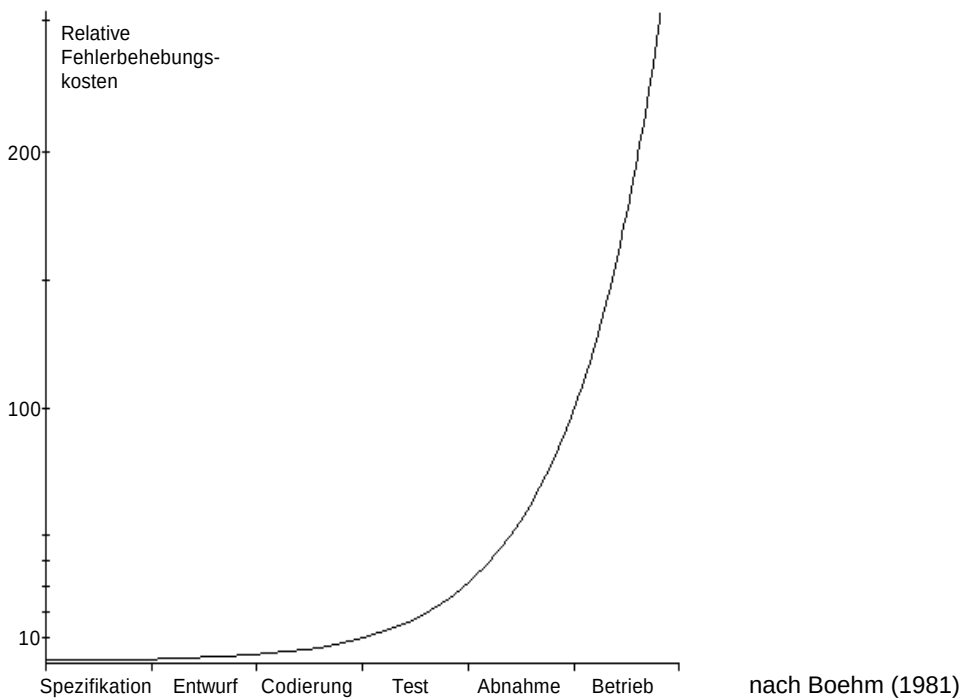


Bild 7.1: Kosten für die Behebung von Fehlern abhängig von ihrer Verweildauer in der Software

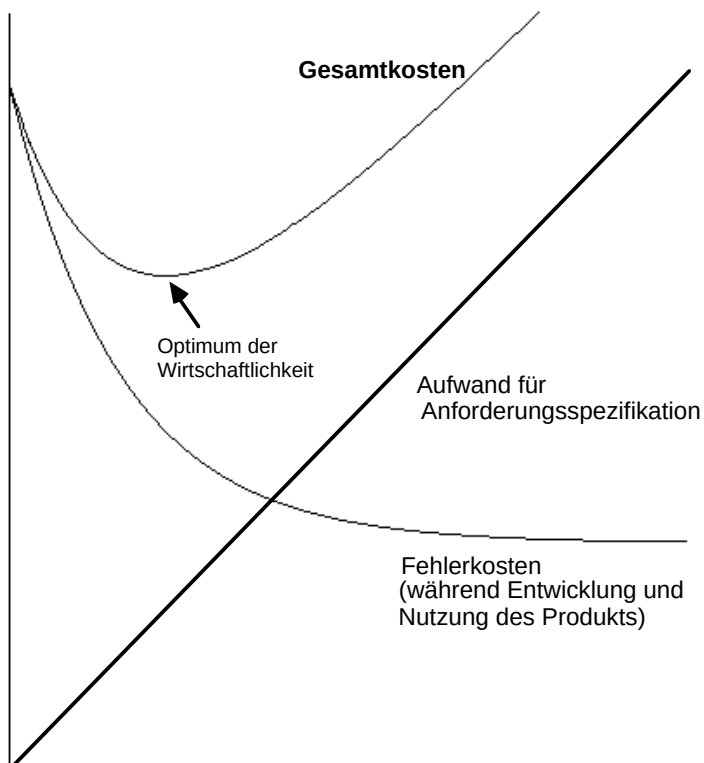


Bild 7.2: Wirtschaftlichkeit der Anforderungsspezifikation

7.1.3 Merkmale einer guten Spezifikation

Um die Fehlerkosten zu senken, müssen wir folglich erreichen, dass (a) wenig Anforderungsfehler gemacht werden und (b) möglichst viele der dennoch gemachten Fehler möglichst früh gefunden werden. Dafür brauchen wir sowohl eine *gute Anforderungsspezifikation* als auch einen *guten Spezifikationsprozess*.

Eine gute Spezifikation zeichnet sich durch folgende Eigenschaften aus:

- Adäquatheit – das beschreiben, was der Kunde will bzw. braucht
- Vollständigkeit – alles beschreiben, was der Kunde will bzw. braucht
- Widerspruchsfreiheit – sonst ist die Spezifikation nicht realisierbar
- Verständlichkeit – für den Kunden und für die Informatiker
- Eindeutigkeit – damit Fehler durch Fehlinterpretationen vermieden werden
- Prüfbarkeit – damit feststellbar ist, ob das realisierte System die Anforderungen erfüllt.

Ein guter Spezifikationsprozess ist charakterisiert durch

- Kundenorientierung
- Methodisches und zielgerichtetes Vorgehen
- Verwendung geeigneter Mittel
- Integration von Erstellung und Prüfung von Anforderungen.

7.2 Der Spezifikationsprozess

Der Prozess des Spezifizierens von Anforderungen umfasst im Wesentlichen drei Aufgaben: die Gewinnung, die Darstellung und die Prüfung der Anforderungen. Der Prozess kann jedoch nicht einfach sequentiell in drei entsprechende Schritte gegliedert werden. Vielmehr muss der Prozess iterativ in enger und ständiger Interaktion zwischen Vertretern des Kunden und den die Spezifikation erstellenden Informatikern/Analytikern ablaufen (Bild 7.3).

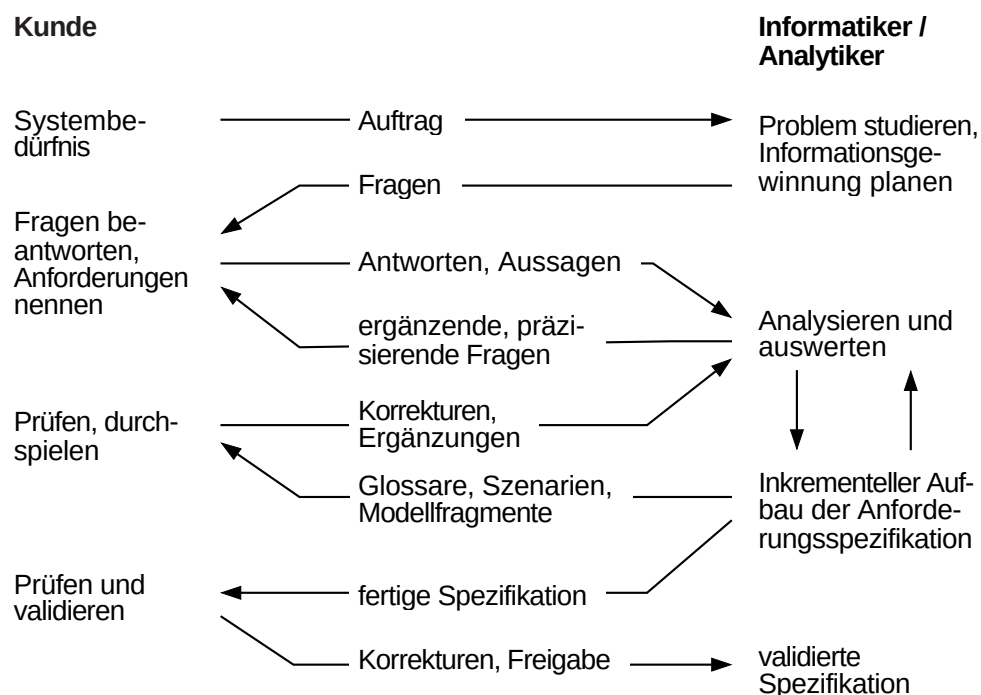


Bild 7.3: Möglicher Ablauf des Spezifikationsprozesses und Interaktionen der Beteiligten

7.3 Dokumentation von Anforderungen

7.3.1 Klassifikation von Anforderungen

Es gibt verschiedene Arten von Anforderungen (Bild 7.4). Zunächst einmal wird zwischen *Projekt-* und *Produktanforderungen* unterschieden. In diesem Kapitel betrachten wir nur letztere. Die *Produktanforderungen* wiederum gliedern sich in funktionale Anforderungen und Attribute. Unter *funktionalen Anforderungen* verstehen wir alle Anforderungen, die sich auf die Funktionalität eines Systems beziehen, das heißt, welche Ergebnisse aufgrund welcher Eingaben zu berechnen und/oder zu liefern sind. Die *Attribute* spezifizieren die Art und Weise, in der diese Funktionalität zu erbringen ist. *Leistungsanforderungen* sind Forderungen bezüglich Zeiten, Mengen, Geschwindigkeiten, Raten, etc. *Besondere Qualitätsanforderungen* sind Forderungen beispielsweise an Zuverlässigkeit oder Benutzerfreundlichkeit. *Randbedingungen* schließlich sind alle Forderungen, welche die Menge der möglichen Lösungen zusätzlich beschränken, z.B. Gesetze und Normen. Leistungsanforderungen, besondere Qualitätsanforderungen und Randbedingungen werden auch *nicht-funktionale* Anforderungen genannt.

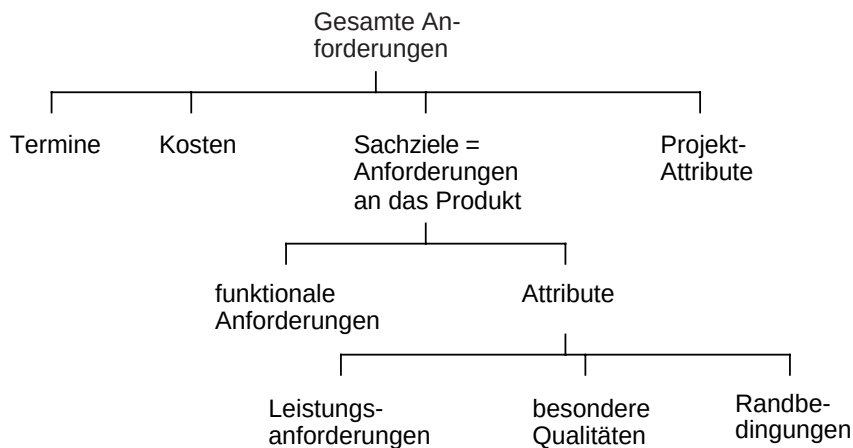


Bild 7.4: Gliederungsbaum der Anforderungen

Zusätzlich müssen die Anforderungen oft nach ihrer Wichtigkeit klassifiziert werden, z.B. in

- *Muss-Anforderungen* – sind unverzichtbar und müssen in jedem Fall erfüllt werden
- *Soll-Anforderungen* – sollten erfüllt werden, sind aber bei zu hohen Kosten verzichtbar
- *Wunsch-Anforderungen* – werden nur erfüllt, wenn dies mit vertretbaren Kosten möglich ist.

Eine solche Klassifikation nach Wichtigkeit ist vor allem in zwei Situationen nötig:

- wenn die Entwicklungskosten eine harte Randbedingung darstellen. Dies ist unter anderem dann der Fall, wenn Software für den Markt entwickelt wird.
- wenn die Anforderungsspezifikation als Grundlage für die Beschaffung eines Systems dient, weil dann die Lösung nicht nach Maß auf die Bedürfnisse zugeschnitten werden kann.

7.3.2 Inhalt und Aufbau einer Anforderungsspezifikation

Eine Anforderungsspezifikation muss *inhaltlich* die folgenden Aspekte abdecken:

- Funktionaler Aspekt
 - Daten: Struktur, Verwendung, Erzeugung, Speicherung, Übertragung, Veränderung
 - Funktionen: Ausgabe, Verarbeitung, Eingabe von Daten
 - Verhalten: Sichtbares dynamisches Systemverhalten, Zusammenspiel der Funktionen

- Fehler: Normalfall und Fehlerfälle
- Leistungsaspekt
 - Datenmengen (durchschnittlich/im Extremfall)
 - Verarbeitungs- /Reaktionsgeschwindigkeit (durchschnittlich/im Extremfall)
 - Verarbeitungszeiten und -intervalle
 - wo immer möglich: *messbare* Angaben!
- Qualitätsaspekt
 - geforderte Qualitäten (z.B. Benutzerfreundlichkeit, Zuverlässigkeit)
- Randbedingungsaspekt
 - einzuhaltende/zu verwendende Schnittstellen
 - Normen und Gesetze
 - Datenschutz, Datensicherung
 - Explizite Vorgaben des Auftraggebers.

Die *Gliederung* des Dokuments und die Art der Darstellung in den einzelnen Kapiteln hängen stark von den verwendeten Methoden und Sprachen ab. Teilweise sind sie auch durch Richtlinien des Kunden oder des entwickelnden Unternehmens oder durch die Anwendung internationaler Normen (z.B. IEEE 830-1993) bestimmt.

Bei der Wahl der Gliederung ist dem Qualitätsmerkmal der Verständlichkeit besonderes Augenmerk zu widmen. Schlechte oder gar nicht vorhandene Gliederungen behindern die Verständlichkeit unter Umständen massiv.

7.4 Gewinnung von Anforderungen

7.4.1 Probleme

Bei der Gewinnung von Anforderungen treten typisch die folgenden Schwierigkeiten auf:

- Unterschiedliche Vertreter des Kunden haben unterschiedliche Vorstellungen über das zu spezifizierende System. Häufig sind schon die Auffassungen und die Begriffsbildung im Anwendungsbereich nicht einheitlich. Requirements Engineering beinhaltet daher auch *Konsensbildung* zwischen divergierenden Vorstellungen der beteiligten Personen.
- Die Kundenvertreter haben zwar eine Vorstellung, was sie wollen, aber sie können ihre Vorstellung nicht formulieren. Manchmal „lösen“ sie dieses Problem, indem sie anstelle ihrer eigentlichen Anforderungen bestehende Lösungen mit vergleichbaren Eigenschaften beschreiben.
- Die Kundenvertreter wissen nicht oder nur sehr vage, was sie eigentlich wollen.

7.4.2 Mögliche Techniken

Die folgenden vier Vorgehensweisen können erfolgreich zur Anforderungsgewinnung eingesetzt werden. Je nach Situation wird eine einzelne Technik oder eine Kombination von Techniken gewählt (Bild 7.5).

- *Begriffe klären* und ein *Glossar* mit Definitionen der wichtigen Begriffe des Anwendungsbereichs erstellen. Damit wird eine begriffliche Grundlage geschaffen, die sicherstellt, dass alle das Gleiche meinen, wenn sie vom Gleichen reden. Ein solches Glossar ist insbesondere dann wichtig, wenn schon die Kundenvertreter untereinander keine klare und einheitliche Begriffswelt haben.

- *Soll-Prozessabläufe untersuchen.* Feststellen, welche äußeren Ereignisse auf das zu spezifizierende System einwirken können und erfragen, wie das System auf welches dieser Ereignisse reagieren soll.
- *Anwendungsszenarien bilden und durchspielen.* Alle Interaktionen der Umgebung (seien dies Menschen oder Sensoren und Stellglieder) mit dem zu spezifizierenden System werden in Form von Szenarien aufgeschrieben und durchgespielt. Szenarien eignen sich besonders gut zur interaktiven Gewinnung und Diskussion von Anforderungen mit den Kunden.
- *Den Anwendungsbereich modellieren.* Feststellen, welche Gegenstände der Anwendung für das zu spezifizierende System relevant sind. (Relevant sind diejenigen Gegenstände, über die das System Information speichern muss, damit es seine Aufgaben erfüllen kann.) Herausfinden, welche Eigenschaften der relevanten Gegenstände und welche Beziehungen der Gegenstände untereinander dem zu spezifizierenden System bekannt sein müssen.

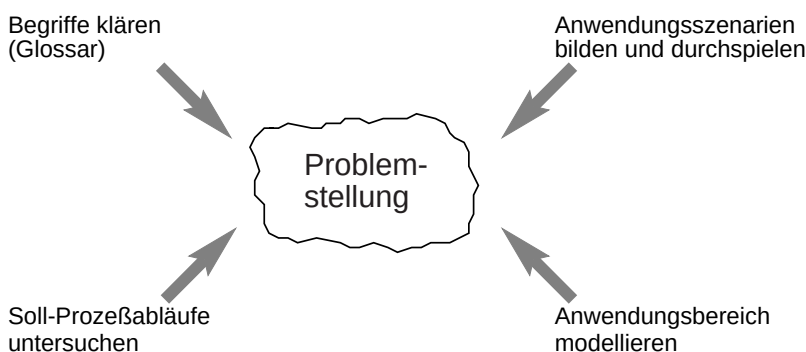


Bild 7.5: Ansätze zur Anforderungsgewinnung

Um die gewünschten Informationen von den Vertretern des Kunden zu erhalten, können verschiedene Techniken eingesetzt werden.

- In *Interviews* werden die Vertreter des Kunden einzeln oder in Gruppen zu höchstens drei Leuten befragt.
- Mit *Fragebogen* können Begriffswelt und Bedürfnisse einer größeren Gruppe von Kundenvertretern erfasst werden.
- In *gemeinsamen Arbeitstagen* (manchmal auch Joint Application Development-Sitzungen genannt) kann eine Gruppe ausgewählter Kundenvertreter und Informatiker gemeinsam die Anforderungen an ein geplantes System erarbeiten.

Die Ergebnisse der Analyse werden mittels geeigneter Methoden und Notationen dargestellt und bilden dann die Anforderungsspezifikation. Anforderungsanalyse und -darstellung laufen iterativ und zeitlich verzahnt ab (Bild 7.3).

7.4.3 Die Rolle der IST-Analyse

In vielen Lehrbüchern, vor allem solchen älteren Datums (klassisch zum Beispiel in McMennamin und Palmer 1984), wird verlangt, dass der Spezifikationsprozess mit einer Analyse des IST-Zustands beginnt. Dabei wird folgendes Vorgehen empfohlen:

- 1 Analyse des IST-Systems; erstellen eines Modells der gegenwärtigen *Implementierung des IST-Systems*.
- 2 Analysieren der dieser Implementierung zugrundeliegenden Konzepte; erstellen des sogenannten *essentiellen Modells des IST-Systems*. Dabei wird von allen implementierungsspezifischen Eigenschaften des IST-Systems abstrahiert.

- 3 Ableiten der Anforderungen an das neue System; erstellen des *essentiellen Modells des SOLL-Systems*. Dieses Modell beschreibt die Anforderungen und ist im Idealfall frei von Entwurfs- und Implementierungsüberlegungen.
- 4 Entwurfs des SOLL-Systems; Erstellen des *Implementierungsmodells des SOLL-Systems*.

Der dritte Schritt in diesem Vorgehen ist der eigentliche Prozess der Anforderungsspezifikation.

Dieses Vorgehen hält jedoch in vielen Fällen einer Wirtschaftlichkeitsprüfung nicht stand. Die Beschäftigung mit dem IST-Zustand ist nur dann gerechtfertigt, wenn

- die Informatiker/Analytiker die IST-Aufgaben und die IST-Arbeitsweise der Benutzer erst kennenlernen und verstehen müssen, damit sinnvolle Gespräche über das SOLL geführt werden können
- die Stärken und Schwächen des IST-Systems nicht bekannt sind
- Teile des IST-Systems übernommen werden sollen und festgestellt werden muss, welche Teile das sind
- das IST-System eins zu eins übernommen werden soll, beispielsweise, wenn ein System auf eine andere Rechnerplattform übertragen werden muss.

Hierzu ist es in der Regel nicht erforderlich, je ein vollständiges Implementierungsmodell und essentielles Modell zu erstellen. Jede über das Notwendige hinausgehende Beschäftigung mit dem IST-System verursacht Kosten, denen kein Nutzen gegenübersteht.

Ferner besteht bei der Ableitung der Anforderungen aus dem IST-System immer die Gefahr, dass alter Wein in neue Schläuche abgefüllt wird und mögliche neue, innovative Wege nicht erkannt werden.

7.5 Darstellung von Anforderungen

Die möglichen Darstellungsformen unterscheiden sich konzeptionell vor allem in drei Aspekten:

- Art der Darstellung: konstruktiv oder deskriptiv
- Formalitätsgrad der Darstellung
- Verfügbare Gliederungs- und Abstraktionsmittel.

Alle drei Aspekte werden im Folgenden skizziert.

7.5.1 Art der Darstellung

Deskriptive Darstellung

Eine deskriptive Darstellung betrachtet das zu spezifizierende System als schwarzen Kasten (Bild 7.6). Die Anforderungen werden durch Beschreibung der geforderten Zusammenhänge zwischen den geforderten Resultatdaten und den gelieferten Eingabedaten dargestellt.



Bild 7.6: Deskriptive Beschreibung von Anforderungen

Beispiele

- Darstellung mit Text in natürlicher Sprache:
«Die Funktion Kontostand liefert den aktuellen Stand des Kontos für die eingegebene Kontonummer.»

- Darstellung in einer formalen Notation:

$$\text{Sqrt: Real} \rightarrow \text{Real}; \text{Pre: } x \geq 0; \text{Post: } |\text{Sqrt}^2(x) - x| < \varepsilon \wedge \varepsilon \leq 10^{-16} \wedge \varepsilon \leq 10^{-6}x.$$

Eine deskriptive Darstellung hat gewichtige Vorteile:

- Die Beschreibung beschränkt sich auf das äußere Verhalten (das heißt: wie das System mit seiner Umgebung interagiert); das ist es, worauf es eigentlich ankommt.
- Die Darstellung ist völlig lösungsneutral.

Diesen Vorteilen stehen jedoch ebenso gewichtige Nachteile gegenüber:

- Eine deskriptive Beschreibung real großer Systeme mit Text ist sehr umfangreich und wenig strukturiert. Die Zusammenhänge zwischen verschiedenen Anforderungen sind oft nicht erkennbar oder gar nicht dargestellt. Eine solche Darstellung ist daher fehlerträchtig und schwierig zu prüfen.
- Eine deskriptive Beschreibung real großer Systeme mit formalen Mitteln ist sehr schwierig und erfordert bestens ausgebildete Spezialisten. Die Prüfung einer solchen Darstellung auf ihre Adäquatheit stellt in vielen Fällen ein fast unüberwindbares Hindernis dar, weil die Vertreter des Kunden die Darstellung weder verstehen noch nachvollziehen können.

Deskriptive Darstellungen sind daher nur für die Darstellung der Anforderungen kleiner, überschaubarer Teilprobleme geeignet. Dort, vor allem bei der Modellierung echter Funktionen, ist ihr Einsatz sinnvoll.

Konstruktive Darstellung

Eine konstruktive Darstellung modelliert das zu spezifizierende System als eine Menge interagierender Komponenten (Bild 7.7). Die Anforderungen werden dargestellt, indem die Erzeugung der geforderten Resultate mit Hilfe der Komponenten und deren Zusammenarbeit in idealisierter Weise beschrieben wird.

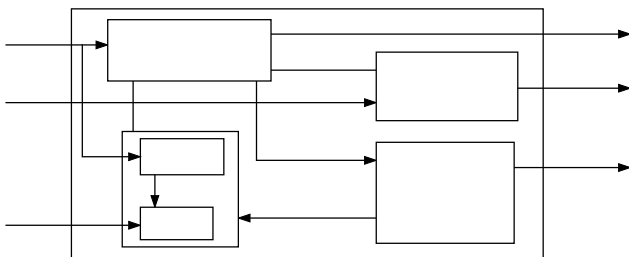


Bild 7.7: Konstruktive Beschreibung von Anforderungen

Beispiel: Das Datenflussdiagramm ist eine typische konstruktive Darstellung. Bild 7.8 zeigt die Spezifikation für die Aufbereitung und Anzeige eines Messwerts.

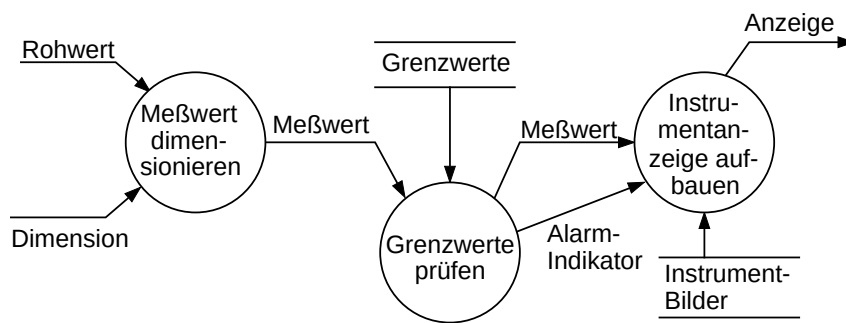


Bild 7.8: Datenflussdiagramm als typischer Vertreter einer konstruktiven Darstellung

Eine konstruktive Darstellung hat die folgenden Vorteile:

- Die Spezifikation ist ein anschauliches Modell der Problemstellung. Sie ist daher leicht verstehbar und nachvollziehbar.
- Die Darstellung ermöglicht die Zerlegung der Gesamtaufgabe in kleinere, besser überschaubare Teilaufgaben.
- Die Kombination von Teilen, die unterschiedlich stark formalisiert sind, ist möglich.
- Das Modell ist eine idealisierte Lösung. In vielen Fällen ist es daher möglich, die Lösung analog zur Struktur der Aufgabenstellung zu strukturieren. Dies spart Entwicklungsaufwand und vereinfacht das Verständnis der Lösungsstruktur.

Die Tatsache, dass das Modell eine idealisierte Lösung ist, stellt aber gleichzeitig auch einen Nachteil dar. Es besteht die Gefahr, dass

- bei der Modellierung auf die Implementierung geschielt wird, statt sich auf die Beschreibung der Anforderungen zu konzentrieren
- suboptimale Lösungen entstehen, wenn die Lösungsstruktur von der Anforderungsstruktur unverändert übernommen wird.

Konstruktive Darstellungen sind vor allem bei der Modellierung von Anforderungen im Großen angezeigt.

7.5.2 Formalitätsgrad der Darstellung

Bei jeder Systementwicklung müssen die vollständig informellen Ideen und Vorstellungen der Auftraggeber in eine vollständig formale Notation (nämlich den Programmcode) überführt werden. Verfahren zur Spezifikation von Anforderungen unterscheiden sich wesentlich in der Frage, wie weit die Formalisierung schon während des Spezifizierens erfolgen soll (Bild 7.9).

Eine *informale Spezifikation* erfolgt in der Regel deskriptiv mit natürlicher Sprache.

Formale Spezifikationen verwenden formale Kalküle, die sich mathematischer Mittel bedienen. Es sind sowohl konstruktive wie deskriptive Verfahren möglich. Petrinetze zum Beispiel sind konstruktiv; Spezifikation mit Logik ist deskriptiv.

Teilformale Spezifikationen basieren auf konstruktiven, anschaulichen Modellen, deren Konstruktionsregeln mit ihren Bedeutungen teilweise formalisiert sind, die aber auch nicht formale Elemente verwenden. Entity-Relationship-Modelle, Zustandsautomaten sowie die verschiedenen Arten der strukturierten und der objektorientierten Analyse sind typische Vertreter dieser Art von Spezifikation.

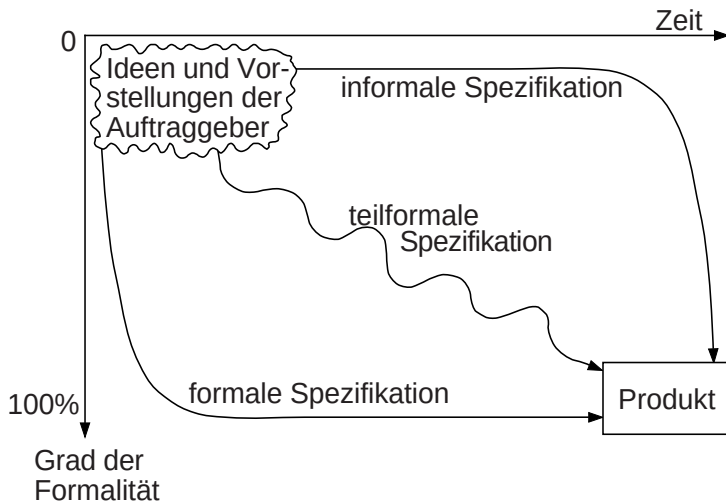


Bild 7.9: Verschiedene Wege bei der Formalisierung

Bei der Beurteilung der Vor- und Nachteile der drei Ansätze sind folgende Überlegungen maßgeblich:

- Informale Spezifikationen haben den großen Vorteil, leicht erstellbar zu sein. Jedoch enthalten große natürlichsprachige Spezifikationen in der Regel sehr viele Unklarheiten, Auslassungen, Mehrdeutigkeiten und Widersprüche. Diese werden beim Prüfen längst nicht alle entdeckt, da sorgfältiges Lesen die einzige zur Verfügung stehende Prüfmethode ist. Damit werden die Fehlerkosten zuwenig gesenkt und die Spezifikation wird unwirtschaftlich.
- Formale Spezifikationen sind immer eindeutig; die Widerspruchsfreiheit ist formal prüfbar. Jedoch ist die Erstellung einer vollständig formalen Spezifikation und ihre Prüfung (vor allem auf Adäquatheit und Vollständigkeit) in der Regel so aufwendig, dass die Kosten im Vergleich zum Nutzen zu hoch sind. Wiederum geht die Wirtschaftlichkeitsrechnung nicht auf.
- Teilformale Darstellungen weisen heute das beste Kosten/Nutzen-Verhältnis auf. Dies gilt insbesondere dann, wenn nicht alle Teile der Spezifikation den gleichen Grad der Formalisierung aufweisen müssen, sondern der Formalitätsgrad dem Entwicklungsrisiko, das mit den einzelnen Komponenten verbunden ist, angepasst werden kann. Teilformale Spezifikationen sind daher heute in den meisten Fällen das Mittel der Wahl.

Beispiel:

Ausschnitte aus der Spezifikation der Steuerung eines Getränkeautomaten

«Der Bediener drückt eine Wahl Taste und bezahlt den geforderten Betrag. Sobald die Summe der eingeworfenen Münzen den geforderten Betrag übersteigt, wird das Getränk zubereitet und ausgegeben. Ferner wird das Wechselgeld berechnet und ausgegeben. Der Bedienvorgang endet, wenn das Getränk entnommen wird und wenn die Bedienung für länger als 45s unterbrochen wird. Mit einer Annulliertaste kann der Bedienvorgang jederzeit abgebrochen werden. Bereits eingeworfenes Geld wird beim Drücken der Annulliertaste zurückgegeben. Nach dem Drücken einer Wahl Taste kann entweder bezahlt oder eine andere Wahl Taste gedrückt werden. Die zuletzt getätigte Auswahl gilt.»

Unklarheit: Reihenfolge von Auswahl und Bezahlung?

Fehler: Was ist bei Betragsgleichheit?

Mehrdeutigkeit: Ist "und" oder "oder" gemeint?

Unklarheit: Abbruch während der Ausgabe des Getränks?

Widerspruch: Die oben geforderte Möglichkeit der Annullierung wird ausgeschlossen

Bild 7.10: Informale Spezifikation und deren Probleme

Sei m_i die i -te eingegebene Münze, $|m_i|$ der Wert dieser Münze, P der geforderte Preis und G die Funktion, welche die Getränkezubereitung auslöst. Dann muss gelten:

$\forall n \in \mathbb{N} \quad \forall m_i, 1 \leq i \leq n$

$$\left[\sum_{i=1}^n |m_i| \geq P \wedge \sum_{i=1}^{n-1} |m_i| < P \right] \leftrightarrow \left[G(m_1 \circ m_2 \circ \dots \circ m_n) \wedge \neg G(m_1 \circ m_2 \circ \dots \circ m_{n-1}) \right]$$

Bild 7.11: Formale Spezifikation des Zusammenhangs zwischen der Eingabe von Münzen und der Auslösung der Zubereitung in der Steuerung eines Getränkeautomaten

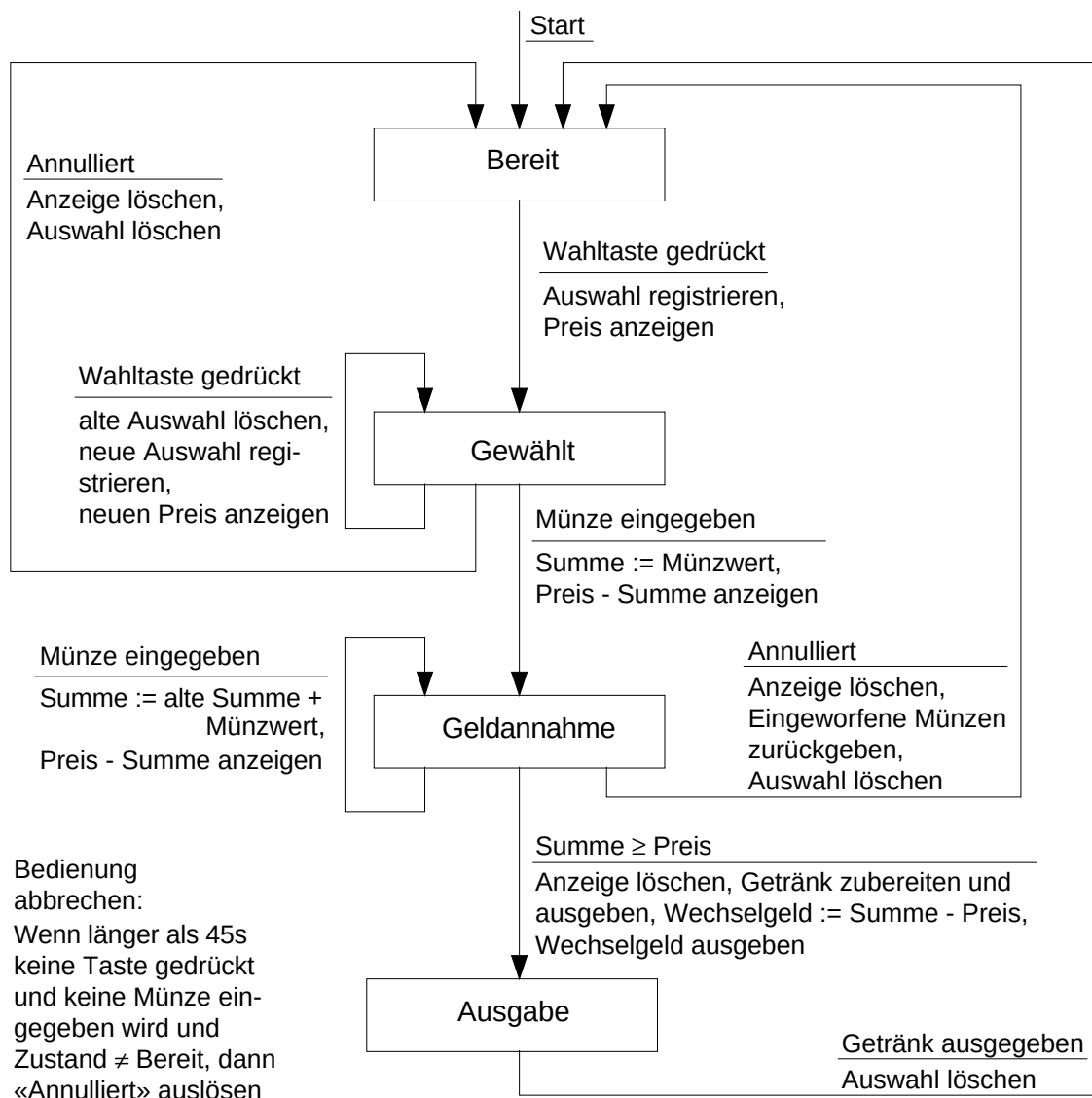


Bild 7.12: Teilformale Spezifikation der Steuerung eines Getränkeautomaten mit einem Zustandsautomaten

7.5.3 Gliederungs- und Abstraktionsmittel

Abstraktionen dienen dazu, ein Modell so zu gliedern und zu strukturieren, dass die Komplexität des Modells besser beherrschbar und die Darstellung damit leichter verständlich wird. Dabei werden drei Abstraktionsarten unterschieden (Bild 7.13).

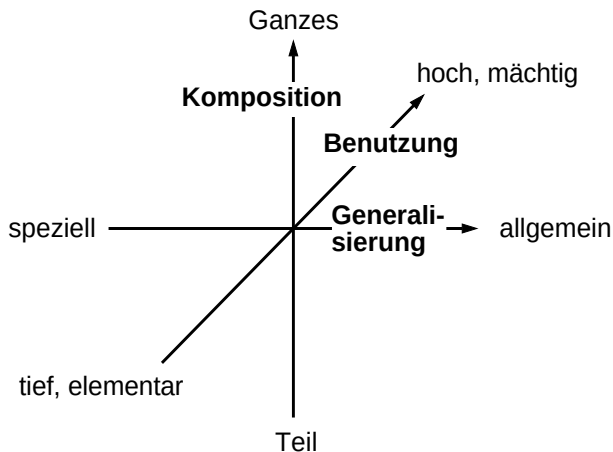


Bild 7.13: Abstraktionsarten und ihre Dimensionen

Die *Komposition* fasst eine Menge von Einzelkomponenten unter Weglassung von Details zu einer übergeordneten Komponente zusammen. Wichtig dabei ist, dass nicht irgendwie zusammengefasst wird, sondern dass logisch zusammengehörende Komponenten so zusammengefasst werden, dass die dabei entstehende übergeordnete Komponente ein in sich möglichst geschlossenes Teilmodell darstellt. Man spricht daher auch von *kapselnder* Komposition. Betrachtet man eine kapselnde Komposition umgekehrt von oben nach unten, stellt sie eine *hierarchische, kapselnde Zerlegung* des Ganzen in Teile dar. Die Kompositionsabstraktion ist ein zentrales Mittel zur Beherrschung der Komplexität umfangreicher Spezifikationen mit vielen Einzelkomponenten (Bild 7.14).

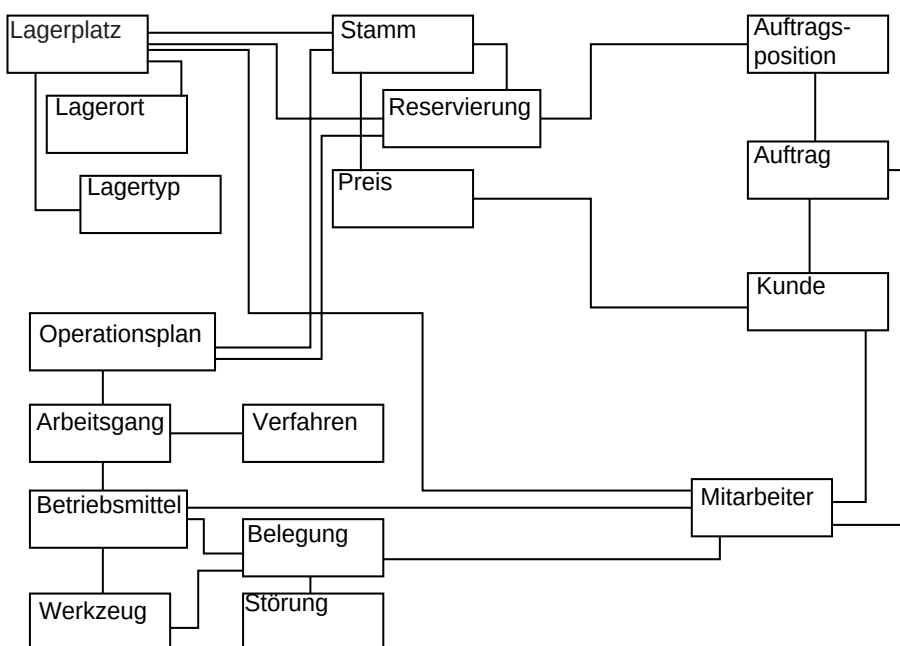


Bild 7.14 a: Kompositionsabstraktion – Komplexe Gebilde mit vielen Komponenten...

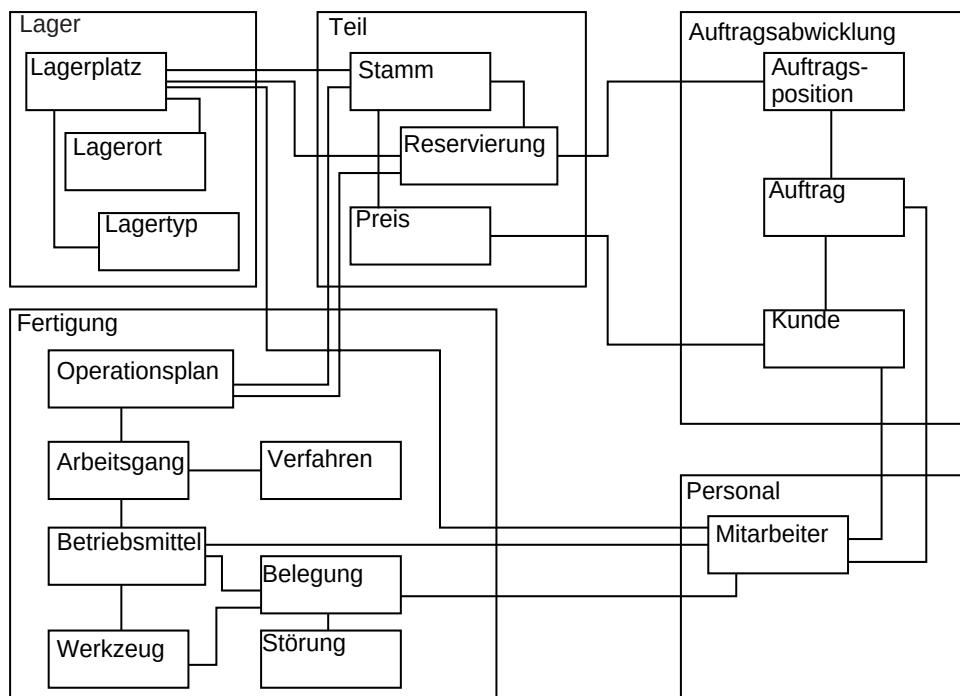


Bild 7.14 b: ...durch Bildung übergeordneter Komponenten...

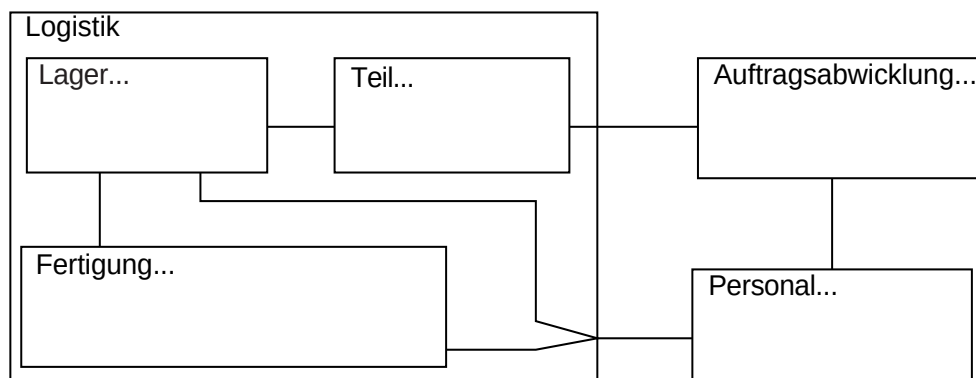


Bild 7.14 c: ...abstrahieren und damit übersichtlicher machen

Die *Benutzung* erlaubt die Bildung von Schichten, deren Komponenten sich einerseits auf Leistungen tieferer Schichten abstützen und andererseits Leistungen für höhere Schichten anbieten. Dabei muss den Benutzern der Leistungen einer Schicht nur bekannt sein, *was* diese Leistungen sind, nicht aber, *wie* das Modell im Inneren der leistungserbringenden Schicht und in allen darunterliegenden Schichten aussieht, d.h. was dort alles im Detail gefordert wird, damit die angebotenen Leistungen zustande kommen. Bei Verwendung konstruktiver Modelle zur Spezifikation ist die Benutzungsabstraktion ein geeignetes Mittel zur Gliederung der Anforderungen in aufeinander aufbauende Schichten (Bild 7.15).

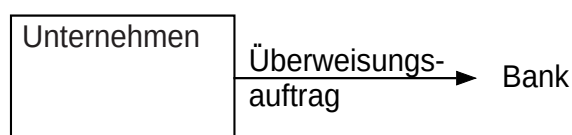


Bild 7.15 a: Benutzungsabstraktion – Komplexe Gebilde...

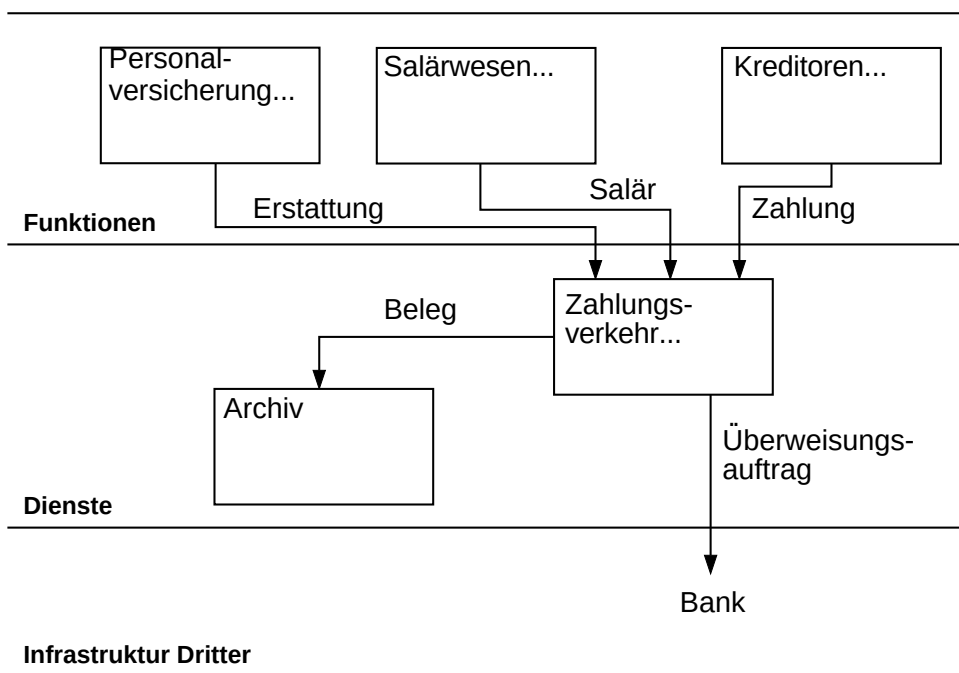


Bild 7.15 b: ... durch Delegieren von Aufgaben und Anordnung der Aufgaben in Schichten übersichtlich und anschaulich modellieren

Die *Generalisierung* fasst die Gemeinsamkeiten einer Menge ähnlicher Komponenten zu einer übergeordneten Komponente zusammen. Dies hilft z.B. bei der Strukturierung der Begriffswelt (Zusammenfassung von Begriffen zu Oberbegriffen) und erlaubt die Modellierung aufgabengerechter Modelle (Bild 7.16).

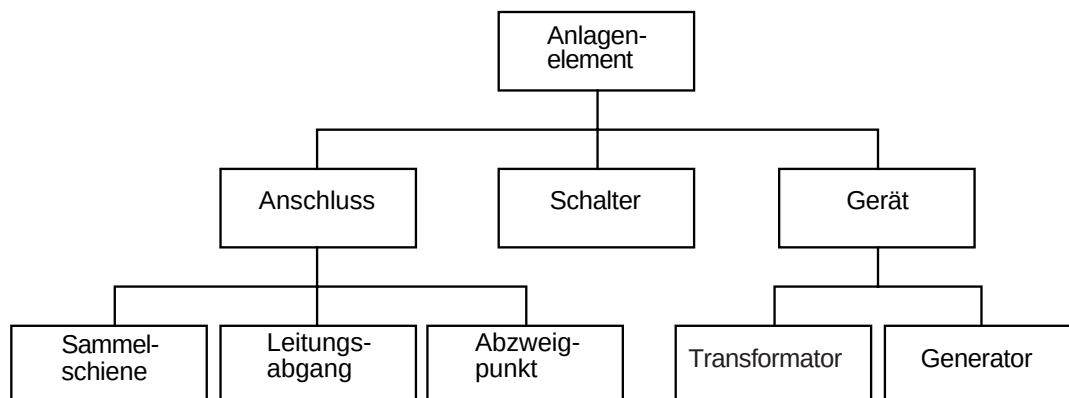


Bild 7.16: Generalisierungsabstraktion – Problemgerechte Modellierung der Begriffswelt der Realität am Beispiel des Engineerings von Unterstationen in Stromverteilungsnetzen

7.5.4 Ausgewählte Spezifikationsmethoden

7.5.4.1 Spezifikation mit natürlicher Sprache

Es gibt keine speziellen Methoden für Spezifikationen mit natürlicher Sprache. In der Regel sind die Anforderungen nummeriert. Ferner werden häufig feste Gliederungsschemata verwendet. Bild 7.10 zeigt ein Beispiel.

7.5.4.2 Algebraische Spezifikation

Algebraische Spezifikation ist eine deskriptive, formale Spezifikationsmethode. Sie wird vor allem für die formale Spezifikation komplexer Datentypen verwendet. Die Grundidee dabei ist, die Wirkung von Operationen auf den Objekten eines Typs durch die Angabe von Axiomen (Ausdrücken, die immer wahr sein müssen) festzulegen. Die Syntax des zu spezifizierenden Datentyps wird durch Angabe von Definitions- und Wertebereichen festgelegt. Das nachfolgende Beispiel (Bild 7.17) zeigt die algebraische Spezifikation eines Stacks (Keller- oder Stapelspeicher, bei dem immer das zuletzt gespeicherte Element wieder entnommen werden kann).

Sei `bool` der Datentyp mit dem Wertebereich `{false, true}` und der Booleschen Algebra als Operationen. Sei ferner `elem` der Datentyp für die Datenelemente, die im spezifizierten Stack zu speichern sind.

TYPE Stack

FUNCTIONS

<code>new:</code>	<code>()</code>	$\rightarrow$	Stack;	-- neuen (leeren) Stack anlegen
<code>push:</code>	<code>(Stack, elem)</code>	$\rightarrow$	Stack;	-- Element hinzufügen
<code>pop:</code>	Stack	$\rightarrow$	Stack;	-- zuletzt hinzugefügtes Element entfernen
<code>top:</code>	Stack	$\rightarrow$	elem;	-- liefert zuletzt hinzugefügtes Element
<code>empty:</code>	Stack	$\rightarrow$	bool;	-- wahr, wenn Stack kein Element enthält
<code>full:</code>	Stack	$\rightarrow$	bool;	-- wahr, wenn Stack voll ist

AXIOMS

$\forall s \in \text{Stack}, e \in \text{elem}$

(1)	$\neg \text{full}(s) \rightarrow \text{pop}(\text{push}(s,e)) = s$	-- Pop hebt den Effekt von Push auf
(2)	$\neg \text{full}(s) \rightarrow \text{top}(\text{push}(s,e)) = e$	-- Top liefert das zuletzt gespeicherte Element
(3)	$\text{empty}(\text{new}) = \text{true}$	-- ein neuer Stack ist leer
(4)	$\neg \text{full}(s) \rightarrow \text{empty}(\text{push}(s,e)) = \text{false}$	-- nach Push ist ein Stack nicht mehr leer
(5)	$\text{full}(\text{new}) = \text{false}$	-- ein neuer Stack ist nicht voll
(6)	$\neg \text{empty}(s) \rightarrow \text{full}(\text{pop}(s)) = \text{false}$	-- nach Pop ist ein Stack niemals voll

Bild 7.17: Algebraische Spezifikation eines Stacks

7.5.4.3 Datenflussorientierte Spezifikation: Strukturierte Analyse

Strukturierte Analyse (DeMarco 1979, McMenamin und Palmer 1984, Yourdon 1989) ist eine teilformale, konstruktive Spezifikationsmethode. Die Anforderungen werden durch eine Hierarchie von *Datenflussdiagrammen* modelliert. Ein Datenflussdiagramm (Bild 7.8) besteht aus Aktivitäten, Datenflüssen und Speichern. Zur Modellierung der Systemumgebung werden außerdem Endknoten verwendet (Bilder 7.18, 7.19). Datenflussdiagramme basieren auf dem Prinzip der *datengesteuerten Verarbeitung*: jede Aktivität arbeitet dann, wenn die von ihr benötigten Datenflüsse eintreffen. Sie erzeugt bei ihrer Arbeit neue Datenflüsse. Diese steuern entweder andere Aktivitäten an oder verlassen das System als Ergebnis.

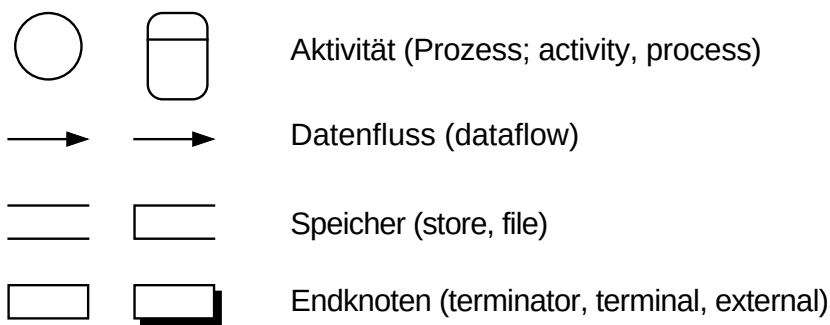


Bild 7.18: Notationen für Datenflussdiagramme

Ein Datenflussdiagramm wird wie folgt interpretiert:

- *Datenflüsse* transportieren Datenpakete, die von Aktivitäten oder Endknoten produziert bzw. konsumiert werden.
- *Aktivitäten* arbeiten nur dann, wenn alle von ihnen benötigten Eingabe-Datenflüsse vorliegen. Die Aktivität konsumiert die Daten, bearbeitet sie und produziert Ausgabe-Datenflüsse. Sie kann dabei zusätzlich Speicherinhalte lesen oder schreiben.
- *Speicher* modellieren Datenbehälter. Ihr Inhalt kann gelesen werden (ohne den Speicher zu verändern) und geschrieben werden (dabei wird der alte Inhalt zerstört).
- *Endknoten* sind Aktivitäten in der Systemumgebung.

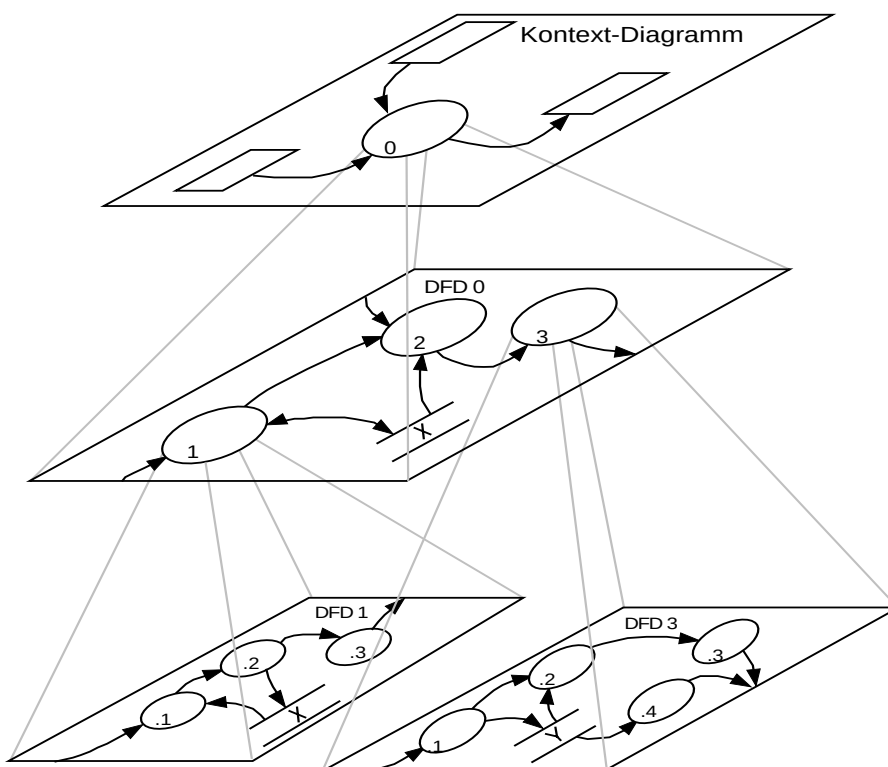


Bild 7.19: Beispiel einer DFD-Hierarchie in Strukturierter Analyse

Datenflussdiagramme sind Übersichtsmodelle. Zur Präzisierung müssen die Namen aller Datenflüsse und Speicher definiert werden. Hierzu dient ein globales *Datenlexikon*, das alle verwendeten Datennamen definiert. Ferner muss die Funktionalität jeder nicht hierarchisch zerlegten Aktivität beschrieben werden. Hierzu dienen sogenannte *Mini-Spezifikationen*, in denen der

zu spezifizierende Zusammenhang zwischen Eingabe- und Ausgabedaten in freier Form beschrieben ist.

Die Datenflussdiagramme eines Systems können hierarchisch in Schichten angeordnet werden. Jede Ebene fasst die Datenflussdiagramme der darunterliegenden Ebenen zu je einer Aktivität zusammen. In der Regel wird ein dazu passendes hierarchisches Nummerierungsschema für Aktivitäten und DFD verwendet. Bei geeigneter Wahl der Zerlegung kann so ein komplexes Modell schrittweise in einfachere Teilmodelle zerlegt werden (Bild 7.19).

7.5.4.4 Verhaltensspezifikation mit Automaten

Verhaltensspezifikationen mit Automaten sind teilformale, konstruktive Modelle. Die Anforderungen werden als eine Menge von Zuständen, Zustandsübergängen, auslösenden Ereignissen und ausgelösten Aktionen beschrieben. Das Modell spezifiziert die Reaktionen des Systems auf eine gegebene Folge äußerer Ereignisse (Bild 7.12).

7.5.4.5 Objektorientierte Spezifikation

Objektorientierte Spezifikationsverfahren sind konstruktive, teilformale Methoden. Es gibt eine Vielzahl verschiedener Ansätze (zum Beispiel Booch (1994), Coad und Yourdon (1991a, b), Jacobson et al. (1992), Rumbaugh et al. (1991), Wirfs-Brock et al. (1990)). Zur Zeit bildet sich mit der Unified Modeling Language UML (Booch, Jacobson, und Rumbaugh 1997) ein Quasi-Standard. Nachstehend werden die Prinzipien der objektorientierten Spezifikation beschrieben.

Die Grundidee ist, ein System durch eine Menge von *Objekten* zu spezifizieren, von denen jedes einen in sich geschlossenen Teil der Daten, der Funktionalität und des Verhaltens des Systems beschreibt. Dabei wird ein Ausschnitt der Realität auf Objekte und deren Eigenschaften abgebildet. Gleichartige Objekte werden durch *Klassen* modelliert.

DEFINITION 7.4. *Objekt (object).* Ein individuell erkennbares, von anderen Objekten eindeutig unterscheidbares Element der Realität.

Beispiel: Die konkrete Person Eva Müller, 36 Jahre alt, Dr. oec. publ., Leiterin Fertigung in der Firma AGP, verheiratet, ein Kind, ... wird als *Objekt* modelliert.

Objekteigenschaften werden als *Attribute* und *Attributwerte*, als *Beziehungen* zu anderen Objekten oder als *Operationen* auf den Objekten modelliert.

Beispiel: Das Objekt EVA MÜLLER hat das Attribut GESCHLECHT mit dem Wert WEIBLICH, die Beziehung LEITET zur ABTEILUNG FERTIGUNG und die Operation BEFÖRDERN.

DEFINITION 7.5. *Klasse (class).* Eine eindeutig benannte Einheit, welche eine Menge gleichartiger Objekte beschreibt.

Beispiel: Die Klasse MITARBEITER mit den Attributen NAME, VORNAME, GESCHLECHT, TITEL, ZIVILSTAND, ANZAHL KINDER, ... und den Beziehungen ARBEITET IN und LEITET zur Klasse ABTEILUNG beschreibt Personen der Art, wie Eva Müller eine ist.

Eine Klasse beschreibt den Aufbau, die Bearbeitungsmöglichkeiten und das mögliche Verhalten von Objekten dieser Klasse. Eine Klassendefinition besteht aus

- der Definition der Attribute der Klasse und der Wertebereiche dazu (lokale Merkmale)
- der Definition der Beziehungen zu anderen Klassen (referenzierte Merkmale)
- der Definition der Operationen, die auf Objekten der Klasse oder auf der Klasse selbst möglich sind

Dabei gibt es immer zwei Sichten für Klassen: In der *intensionalen* Sicht ist eine Klasse ein *Typ*. Alle Objekte der Klasse sind nach diesem Typ aufgebaut. In der *extensionalen* Sicht dagegen ist eine Klasse eine *Menge von Objekten*. Häufig werden beide Sichten miteinander vermischt.

Klassen stehen wie folgt in Beziehung zueinander:

- *Assoziation*: Die Objekte einer Klasse sind Merkmale von Objekten einer anderen Klasse. Gilt in der Regel auch umgekehrt, d.h. Assoziationen sind bidirektional.
- *Benutzung*: Die Objekte einer Klasse benutzen Attribute oder Operationen einer anderen Klasse zur Bereitstellung ihrer eigenen Attribute und Operationen.
- *Vererbung*: Eine Klasse ist Unterklasse einer anderen Klasse. Sie erbt alle Attribute und Operationen dieser Klasse, d.h. alle Objekte der Klasse verfügen über diese, ohne dass sie in der Klasse lokal definiert worden wären.
- Zu den Assoziationen werden *Kardinalitäten* modelliert: wieviele Objekte der assoziierten Klasse müssen mindestens / dürfen höchstens mit einem Objekt der eigenen Klasse assoziiert sein.

```

KLASSE Mitarbeiter im Monatslohn
UNTERKLASSE von Mitarbeiter
ATTRIBUTE (Name, Kardinalität, Wertebereich)
  Leistungslohnanteil      [1..1]    CHF
  Überzeitsaldo            [1..1]    Stunde
  Ferienguthaben           [1..1]    Tag
  ...
BEZIEHUNGEN (Name, Kardinalität, mit Klasse)
  eingestuft in            [1,1]     Lohnklasse
OPERATIONEN
  Lohn zahlen
    Voraussetzung: Mitarbeiter ist aktiv
    Ergebniszusicherung: Zahlungsauftrag zugunsten des Mitarbeiters ist erteilt mit Grundlohn aus
    Lohnklasse und Leistungslohnanteil
BENUTZT (Klasse.Operation)
  Zahlungsauftrag.Erteilen
  
```

Bild 7.20: Beispiel einer Klassendefinition am Beispiel einer Klasse aus einem Personalinformationssystem

Hinweis: Da MITARBEITER IM MONATSLOHN eine Unterklasse von MITARBEITER ist, werden alle Attribute (zum Beispiel NAME und VORNAME), Beziehungen und Operationen der Klasse MITARBEITER automatisch übernommen, ohne dass sie nochmals definiert werden müssen.

Es gibt heute eine Vielzahl verschiedener *Notationen* für Klassenmodelle. Die meisten Notationen verwenden Rechtecke für Klassen und Linien für Assoziationen. Die Klassensymbole werden häufig durch zwei Linien unterteilt: oben steht der Klassenname, in der Mitte die wichtigsten Attribute, unten die wichtigsten Operationen. Die Notation für Benutzung und Vererbung ist sehr uneinheitlich. Hier werden verwendet: Pfeile für Benutzung, gegabelte Linien für Vererbung. Für die Notation von Kardinalitäten gibt es eine verwirrende Fülle von graphischen und numerischen Notationssystemen. In Bild 7.21 wird die Notation der UML (Unified Modeling Language, Booch, Jacobson, und Rumbaugh 1997) verwendet. Die Kardinalitäten sind in Beziehungsrichtung zu lesen, zum Beispiel "Eine ABTEILUNG BESCHÄFTIGT mindestens einen und höchstens beliebig viele MITARBEITER" (ein Stern steht für beliebig viele).

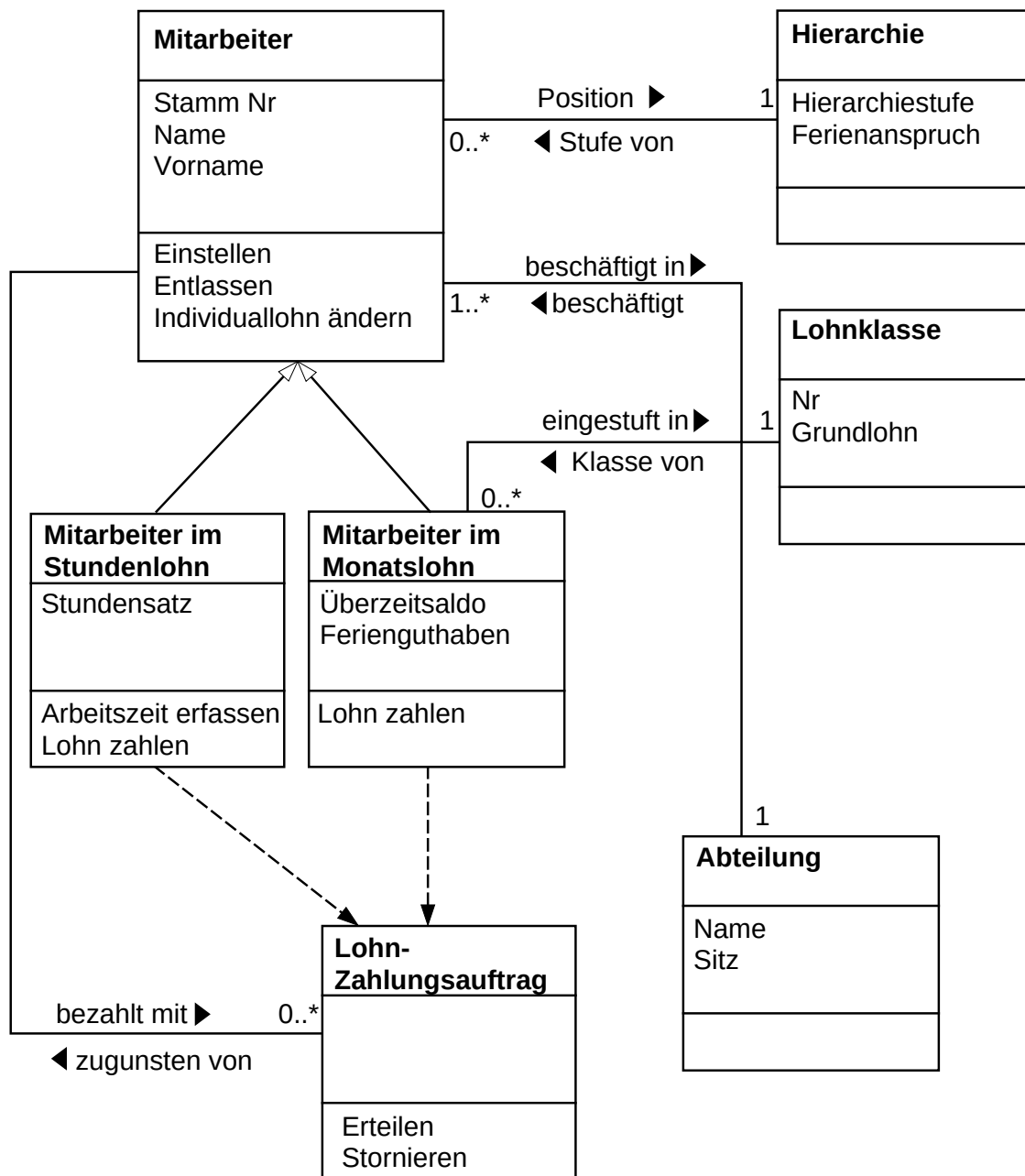


Bild 7.21 Beispiel eines Klassenmodells (stark vereinfachtes Modell eines Personalinformationssystems)

Als *Alternative* oder als *Ergänzung* zu Klassenmodellen können *Objektmodelle* verwendet werden. In solchen Modellen werden keine konkreten Objekte (d.h. solche mit bekannter Identität und bekannten Attributwerten) modelliert, sondern *abstrakte Objekte*, die als Muster bzw. als Repräsentanten für konkrete Objekte stehen. In den meisten heute verwendeten Ansätzen werden Objektmodelle entweder *gar nicht verwendet* oder sie dienen in *Ergänzung* zu einem Klassenmodell zur Modellierung des Arbeitskontextes der Objekte einer Klasse. Objektmodelle *anstelle* von Klassenmodellen haben große Vorteile, wenn verschiedene Objekte der gleichen Klasse zu modellieren sind, und wenn ein Modell hierarchisch in Komponenten zerlegt werden soll. Dagegen ist die Modellierung von Assoziationen und Vererbungsbeziehungen in Objektmodellen umständlicher als in Klassenmodellen.

Bild 7.22 zeigt eine Situation, in der ein Objektmodell einem Klassenmodell in der Ausdruckskraft überlegen ist. In manchen Klassenmodellen, so auch in UML, behilft man sich,

indem die Beziehung zusätzlich mit den *Rollen*, welche die in Beziehung gesetzten Objekte spielen (d.h. hier TUTOR und ERSTSEMESTER) beschriftet werden.

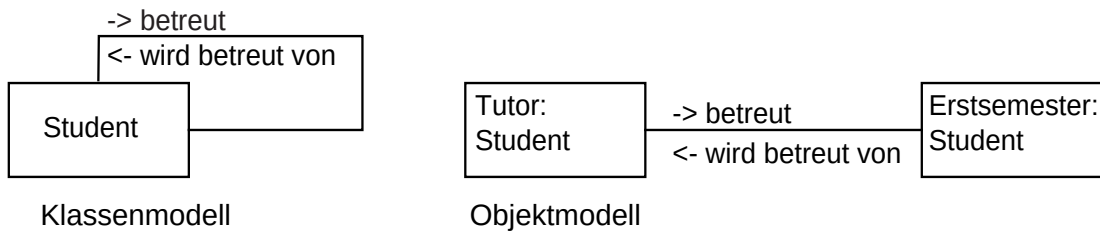


Bild 7.22: Klassenmodell vs. Objektmodell

7.5.4.6 Spezifikation mit Anwendungsfällen

Die Klassenmodelle in Objektorientierten Spezifikationen modellieren weder den Systemkontext noch die Interaktionen zwischen einem System und seiner Umgebung. Weil dies aber beides wichtige Elemente einer Anforderungsspezifikation sind, werden objektorientierte Spezifikationen heute meist durch ein *Anwendungsfallmodell*, welches die Interaktionen zwischen den systemexternen *Akteuren* und dem System modelliert, ergänzt (Jacobson et al. 1992).

Definition 7.6. *Anwendungsfall.* Eine durch genau einen Akteur angestoßene Folge von Systemereignissen, welche für den Akteur ein Ergebnis produziert und an welchem weitere Akteure teilnehmen können.

Jeder Anwendungsfall spezifiziert die notwendige Folge von Interaktionschritten bei der Ausführung einer essentiellen Systemfunktion. Er wird informal durch Text oder teilformal, beispielsweise durch Zustandsautomaten (vgl. Bild 7.12), modelliert. Ein Anwendungsfalldiagramm zeigt den Systemkontext und die Menge aller Anwendungsfälle (Bild 7.23). Die zwei «uses»-Beziehungen modellieren die gemeinsame Verwendung eines Sub-Anwendungsfalls durch zwei andere Anwendungsfälle.

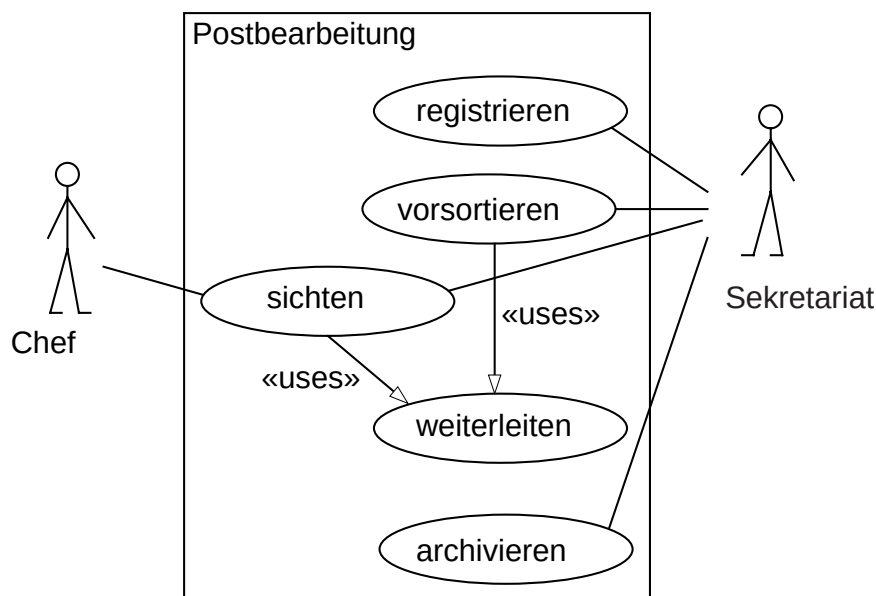


Bild 7.23: Anwendungsfalldiagramm

7.6 Prüfung von Anforderungen

7.6.1 Worum es geht

Beim Prüfen einer Anforderungsspezifikation geht es darum, Abweichungen von der geforderten Qualität der Spezifikation festzustellen. Mit anderen Worten, bei der Prüfung sollen möglichst viele Fehler, Lücken, Unklarheiten, Mehrdeutigkeiten, etc. gefunden (und anschließend behoben) werden.

Dabei haben nicht alle Qualitätsmerkmale das gleiche Gewicht. In erster Priorität sollte auf Adäquatheit, Vollständigkeit, Widerspruchsfreiheit und Verständlichkeit geprüft werden, in zweiter Priorität auf Prüfbarkeit und Eindeutigkeit und in dritter Priorität auf alle übrigen Merkmale.

Die Prüfung einer Spezifikation auf Adäquatheit, Vollständigkeit und Widerspruchsfreiheit wird auch *Validierung* genannt.

7.6.2 Beteiligte

Die Prüfung erfolgt sinnvollerweise unter Federführung von Informatikern, die insbesondere die verwendete Darstellung ohne fremde Hilfe interpretieren können. Zusätzlich müssen aber die Vertreter des Kunden zwingend in den Prüfprozess einbezogen werden, denn nur sie können schlussendlich die Adäquatheit und Vollständigkeit der Spezifikation beurteilen.

7.6.3 Zeitpunkt der Prüfung

Die abschließende Prüfung einer Spezifikation erfolgt zu einem Zeitpunkt, wo

- die Spezifikation fertig ist
- aber noch genug Zeit bleibt, die gefundenen Mängel zu beheben.

Bei umfangreichen Spezifikationen sind zusätzlich Zwischenprüfungen begleitend zur Erstellung der Spezifikation erforderlich.

7.6.4 Prüfverfahren

Für die Prüfung einer Anforderungsspezifikation kommen vier Verfahren in Betracht

- Reviews
- Prüf- und Analysemittel in Werkzeugen
- Simulation
- Prototypen.

Ein *Review* ist eine formell organisierte Zusammenkunft von Personen zur inhaltlichen oder formellen Überprüfung eines Produktteils (Dokument, Programmstück, etc.) nach vorgegebenen Prüfkriterien (vgl. Kapitel 9.4). Reviews sind *das* Mittel zur Prüfung von Dokumenten.

Prüf- und Analysemittel in Werkzeugen werden immer dann eingesetzt, wenn eine Spezifikation mit Hilfe von Werkzeugen erstellt wurde. Insbesondere Lücken und Widersprüche in der Spezifikation lassen sich mit solchen Prüfverfahren finden. Beispielsweise kann ein Werkzeug prüfen, ob jeder verwendete Datenname auch irgendwo definiert ist.

Mit einer *Simulation* kann die Adäquatheit des Verhaltens des spezifizierten Systems in bestimmten Situationen untersucht werden.

Ein *Prototyp* ist ein lauffähiges Stück Software, welches Teile eines zu entwickelnden Systems vorab realisiert (vgl. Kapitel 3.2.5). Er dient als Modell für die weitere Entwicklung oder für das zu schaffende Produkt. Ein Prototyp ist das mächtigste verfügbare Mittel für die Beurteilung der Adäquatheit einer Spezifikation durch die zukünftigen Benutzer. Er ermöglicht

in beschränktem Rahmen eine Erprobung des gewünschten Systems in seiner geplanten Einsatzumgebung. Aufgrund der im Vergleich zu anderen Prüfverfahren hohen Kosten für einen Prototyp muss in jedem Einzelfall entschieden werden, wie weit die Entwicklung von Prototypen für die Prüfung von Anforderungen aufgrund des Entwicklungsrisikos notwendig ist.

Aufgaben

- 7.1 Welches sind die Eigenschaften einer guten Spezifikation? Begründen Sie ihre Aussagen.
- 7.2 Nehmen Sie an, Sie bräuchten ein Zutrittskontrollsystem (Schlüsselkarten o.ä.) für die Haustür eines Bürogebäudes. Beschreiben Sie ihre Vorstellungen über das Verhalten dieses Systems mit Szenarien.
- 7.3 Spezifizieren Sie einen Taschenrechner Ihrer Wahl in natürlicher Sprache. Gliedern Sie die Anforderungen gemäß der Struktur der „Anforderungen an das Produkt“ in Bild 7.4.

Ergänzende und vertiefende Literatur

Davis (1993) ist ein Lehrbuch über Requirements Engineering, das vor allem die klassischen Techniken beschreibt.

Gause und Weinberg (1989) beleuchten primär den Aspekt der Anforderungsgewinnung und der sozialen Dimension des Requirements Engineerings.

Kotonya und Sommerville (1998) ist eine Monographie über Requirements Engineering.

Oestereich (1998) gibt eine sehr gute Einführung in die UML.

Verschiedene objektorientierte Spezifikationsverfahren sind beschrieben in: Booch (1994), Booch, Jacobson und Rumbaugh (1997), Coad und Yourdon (1991a), Jacobson et al. (1991), Rumbaugh et al. (1991).

Die folgenden Autoren behandeln klassische Spezifikationsverfahren. Information Engineering: Martin (1990); SSADM: Ashworth und Goodland (1990); Strukturierte Analyse: DeMarco (1979), McMenamin und Palmer (1984) sowie Yourdon (1989)

McDermid (1991) beschreibt verschiedene, teilweise weniger bekannte klassische Spezifikationsverfahren sowie Verfahren zur formalen Spezifikation (Z, VDM, algebraische Spezifikation).

Pepper et al. (1982) geben eine Einführung in die algebraische Spezifikation.

Zitierte Literatur

Ashworth, C., M. Goodland (1990). *SSADM: a Practical Approach*. McGraw-Hill London, etc.

Boehm, B. (1981). *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, N.J.

Booch, G. (1994). *Object Oriented Analysis and Design with Applications*. Second Edition. Redwood City, Ca.: Benjamin/Cummings.

Booch, G., Jacobson, J., Rumbaugh, J. (1997). *The Unified Modeling Language for Object-Oriented Development, Documentation Set 1.1*. Rational Software Corporation.

Coad, P., E. Yourdon (1991a). *Object-Oriented Analysis*. 2nd edition. Englewood Cliffs, N.J.: Prentice-Hall. [auf Deutsch: Objektorientierte Analyse, 1994 im gleichen Verlag]

- Coad, P., E. Yourdon (1991b). *Object-Oriented Design*. Englewood Cliffs, N.J.: Prentice-Hall. [auf Deutsch: Objektorientiertes Design, 1994 im gleichen Verlag]
- Davis, A.M. (1993). *Software Requirements: Objects, Functions and States*. Englewood-Cliffs, N.J.: Prentice-Hall.
- DeMarco, T. (1979). *Structured Analysis and System Specification*. Yourdon Press, Prentice-Hall, Englewood Cliffs, N.J.
- Gause, D.C., G.M. Weinberg (1989). *Exploring Requirements: Quality before Design*. Dorset House, New York. [auf Deutsch: Software Requirements: Anforderungen erkennen, verstehen und erfüllen. Hanser, München, 1993.]
- IEEE (1990). *Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990. IEEE Computer Society Press.
- IEEE (1993). *IEEE Recommended Practice for Software Requirements Specifications*. IEEE Standard 830-1993. IEEE Computer Society Press.
- Jacobson, I., M. Christerson, P. Jonsson, G. Övergaard (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Amsterdam; Reading, Mass. [u.a.]: Addison-Wesley.
- Kotonya, G., I. Sommerville (1998). *Requirements Engineering: Processes and Techniques*. Chichester, etc.: John Wiley & Sons.
- Martin, J. (1990). *Information Engineering. Book 2: Planning and Analysis*. Prentice-Hall, Englewood Cliffs, N.J.
- McDermid, J. (1991). *Software Engineer's Reference Book*. Oxford: Butterworth-Heinemann. ca. 1200 p.
- McMenamin, S.M., J.F. Palmer (1984). *Essential Systems Analysis*. Yourdon Press, New York.
- Oestereich, B. (1998). *Objektorientierte Softwareentwicklung*. R. Oldenbourg Verlag München.
- Pepper, P. et al. (1982). Abstrakte Datentypen: Die algebraische Spezifikation von Rechenstrukturen. *Informatik-Spektrum* 5, (1982). 107-119.
- Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, W. Lorensen (1991). *Object-Oriented Modeling and Design*. Englewood Cliffs, N.J.: Prentice-Hall. [auf Deutsch: Objektorientiertes Modellieren und Entwerfen, München: Hanser, 1993]
- Wirfs-Brock, R., B. Wilkerson, L. Wiener (1990). *Designing Object-Oriented Software*. Englewood Cliffs, N.J.: Prentice-Hall. [auf Deutsch: Objektorientiertes Software Design, München: Hanser, 1993]
- Yourdon, E. (1989). *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs, N.J. 672 p.

8. Realisierung

Das Thema Realisierung wird hier nur summarisch behandelt, da es in der Programmierausbildung bereits vertieft worden ist. Inspektions- und Testverfahren werden im Kapitel 9 behandelt.

8.1 Möglichkeiten der Realisierung

Ein Lösungskonzept (d.h. eine Systemarchitektur) lässt sich auf verschiedene Arten realisieren:

- durch Entwerfen und Codieren
- durch Entwerfen und Generieren
- durch Konfigurieren vorhandener Komponenten.

In allen Fällen müssen die realisierten Einzelkomponenten zu einem System integriert werden, und müssen die Komponenten wie auch das System geprüft werden, um mögliche Fehler zu finden und sie anschließend zu beheben. Sowohl beim Entwerfen wie beim Codieren werden wo möglich vorhandene Komponenten und vorhandene Ideen (zum Beispiel Entwurfsmuster) genutzt.

Im klassischen Detailentwurf werden die zu codierenden Algorithmen und Datenstrukturen festgelegt. Dort, wo das Lösungskonzept noch zuwenig detailliert ist, werden die nötigen Details erarbeitet. In der Codierung wird dieser Entwurf in einer Programmiersprache ausformuliert.

Dort, wo der Code generiert werden kann bzw. soll, entsteht ein formelles Entwurfsdokument, dessen Syntax den Vorschriften des Generators genügen muss. Typische Kandidaten für die Generierung sind Datenbanken und einfache Datenbank Anwendungen einschließlich der zugehörigen Bildschirmmasken.

Aus hinreichend präzisen teilformalen Entwurfsmodellen können mit geeigneten Werkzeugen *Coderahmen* generiert werden, die dann von Hand auszuprogrammieren sind.

8.2 Systematisches Programmieren

Die Qualität der resultierenden Software wie auch der für die Prüfung erforderliche Aufwand lassen sich durch systematisches Programmieren erheblich beeinflussen. Besonders wichtige Punkte sind

- die geeignete Wahl von Algorithmen und Datenstrukturen und deren adäquate Programmierung
- die Verwendung geschlossener Ablaufkonstrukte (solche mit einem Eingang und einem Ausgang)
- die saubere Gliederung der Software in Prozeduren und Module
- die Verwendung symbolischer Konstanten
- die Verwendung selbstdokumentierender Namen für Prozeduren, Variablen, Konstanten und Typen
- die Dokumentation aller Definitionen durch Kommentare
 - bei Prozeduren: Voraussetzungen, die beim Aufruf erfüllt sein müssen, Ergebniszusicherungen (Aussagen, die nach der Ausführung der Prozedur gelten), Bedeutung und Kommunikationsrichtung der Parameter

- bei Variablen und Konstanten: Bedeutung der Variable bzw. Konstante im Kontext des Programms
- defensives Programmieren
- Vermeiden von Trickprogrammierung und Programmierung mit Seiteneffekten.

Häufig werden Vorschriften zur Programmiersystematik und zum Programmierstil in Entwicklungsrichtlinien festgehalten.

Beispiel 8.1 zeigt ein Beispiel für ein nach den obenstehenden Kriterien erstelltes Programm. Beispiel 8.2 demonstriert, wie man es nicht machen sollte. Das Java-Programm in Beispiel 1 wurde nach einer Entwicklungsrichtlinie für Java (Berner et al. 1996) dokumentiert.

Beispiel 8.1: Systematisches Programmieren (in Java)

Dargestellt ist ein Auszug aus einer Klasse zur Verwaltung von Messreihen. Da die Messreihen unterschiedlich lang sein können, sind sie intern als verkettete Listen von Datenobjekten realisiert. Nach außen ist jedoch für jede Messreihe nur ein Objekt sichtbar, das mit den von der Klasse bereitgestellten Operationen manipuliert wird.

```
/* -----
AUTOR:      Martin Arnold
PROJEKT:    Beispiel 8.1, Vorlesung Software Engineering I
JAVA:       Symantec Café, Projectmanager 8.2b3
COPYRIGHT:  Forschungsgruppe Requirements Engineering,
            Institut für Informatik, Universität Zürich
KLASSE:     MeasurementList
VERSION:    1.01 von M. Glinz am 18.12.1996
-----

ZWECK
Die Objekte der Klasse MeasurementList modellieren Messreihen.

VERANTWORTLICHKEITEN
Speicherung, Skalierung, Filterung und Abfrage von Messreihen.
*/
import java.awt.List;

public class MeasurementList extends List    // MeasurementList wird Unterklasse von List
{

/* --- KLASSEN-Variablen ---*/
    final private static double TOLERANZ = 0.30; // 30% zugelassene TOLERANZ nach unten

/* --- INSTANZ-Variablen ---*/
    double        wert;           // Listeneintrag: Messwert als reelle Zahl
    MeasurementList naechstes;    // "Zeiger" auf nächstes Element der verketteten Liste

/* Hinweis zur Dokumentation: Ein Objekt, auf das eine Methode (Operation) angewendet
wird, wird aktuelles Objekt genannt.*/
/* Hinweis zur Implementierung: Jede (auch die leere) Liste wird durch ein Objekt
abgeschlossen, dessen Wert undefiniert ist und dessen Zeiger (naechstes) auf kein
weiteres Objekt, sondern auf "null" zeigt.*/

void add(double element)
    /* Erste Version von Martin Arnold, 06.12.1996
       Letzte Aenderung von Martin Glinz, 18.12.1996
```

```

    POST    An die aktuelle Liste (d.h. diejenige, die mit dem aktuellen Objekt beginnt)
            wird ein Element mit der Zahl 'element' als Wert angehängt. Das aktuelle
            Objekt bleibt erstes Element der Liste.

    /*
    {
    MeasurementList temp = this;    // Referenz auf aktuellen Listenbeginn (erstes Element)

    while ( temp.naechstes != null ) // Suche nach letztem Element
        { temp = temp.naechstes; }

    temp.naechstes = new MeasurementList();    // Zuweisung neuer Listenzeiger
    temp.wert = element;    // Zuweisung des einzufügenden Wertes
    }

    /*
    Weitere Methoden; in diesem Beispiel weggelassen ...
    */

MeasurementList ausreisserEntfernen(double referenzwert)
    /*    Erste Version in Modula-2 von Martin Glinz, 1993
        Reimplementiert in JAVA von Martin Arnold, 6.12.1996
        Letzte Version von Martin Glinz, 18.12.1996

    PRE        Liste ist ordnungsgemäss verkettet. Beim letzten Element ist der Zeiger
                naechstes = null.

    POST        Die aktuelle Liste ist wie folgt modifiziert: Alle Listenelemente, deren Wert um
                mehr als TOLERANZ (in Prozent) kleiner als der Referenzwert ist, sind aus der
                Liste entfernt. Als Funktionswert wird das erste Listenelement (Listenanker) der
                bereinigten Liste zurückgegeben.

    /*
    {
    MeasurementList temp = this; // Referenz auf aktuellen Listenbeginn (erstes Element)
    MeasurementList vorgaenger = new MeasurementList(); //    Initialisierung eines
                                                                Vorgaengerelements
    MeasurementList listenanker = temp; // Listenanker = erstes Listenelement

    /*    Die Liste wird hier vollständig durchlaufen und sämtliche
        Elemente unterhalb der TOLERANZgrenze werden entfernt.*/
    while ( temp.naechstes != null ) // Liste durchlaufen bis zum letzten Eintrag
        {
            if (temp.wert < (referenzwert * (1.0 - TOLERANZ)))
                { /* Element unterhalb der TOLERANZgrenze */
                    if (vorgaenger.naechstes != null)
                        { /* aktuelles Element ist nicht erstes Listenelement (Listenanker) */
                            vorgaenger.naechstes = temp.naechstes; // Element ausketten
                            temp = vorgaenger.naechstes;
                        }
                    else
                        { /* aktuelles Element ist erstes Listenelement (Listenanker) */
                            listenanker = temp.naechstes; // Listenanker neu setzen
                            temp = listenanker;
                        }
                }
            else
                { /* Element im TOLERANZbereich */
                    vorgaenger = temp;
                    temp = temp.naechstes;
                }
        }
    }
    }
    
```

```

        }
    }
    return listenanker; // Rückgabe: Erstes Listenelement der bereinigten Liste
}

```

Beispiel 8.2: Trick-Programmierung mit Seiteneffekten (in C)

Die nachfolgende Prozedur hängt die Zeichenkette *s* an die Zeichenkette *t* an und speichert das Ergebnis in *t*. Man beachte, dass die Rümpfe der beiden verwendeten Schleifen leer sind. Alle Effekte sind als Seiteneffekte in den Schleifensteuerungen versteckt.

```

void strcat (char *s, char *t)
{
    WHILE (*t++); FOR (t--;*t++=*s++); /* t <- t concat s */
}

```

8.3 Prüfung und Integration

Software wird typischerweise komponentenweise realisiert. Jede Komponente wird nach ihrer Fertigstellung zunächst für sich allein *geprüft*. Die wichtigsten Prüfverfahren sind:

- Prüfung auf syntaktische Richtigkeit durch einen Compiler
- Prüfung auf inhaltliche Richtigkeit durch Inspektion
 - Selbstinspektion durch die Autorin oder den Autor
 - Formale Inspektion (vgl. Kapitel über Qualitätsmanagement) durch Dritte
- Komponententest (vgl. Kapitel über Qualitätsmanagement).

Die fertiggestellten und geprüften Komponenten müssen anschließend zu einem System *integriert* werden. Die Integration erfolgt schrittweise. Jeder Integrationsschritt ist mit einem zugehörigen *Integrationstest* verbunden, der die Richtigkeit des Integrationsschritts sichert.

Nach Abschluss der Integration wird das fertiggestellte System einem gründlichen *Systemtest* unterworfen. Im Systemtest sollen möglichst viele derjenigen Fehler gefunden werden, die bei den Komponenten- und Integrationstests übersehen wurden, weil sie nur beim Zusammenspiel zwischen verschiedenen Komponenten auftreten.

Je nach Vertragsverhältnis zwischen Software-Ersteller und Auftraggeber erfolgt anschließend noch eine formale *Abnahme*. Dabei werden vorher vereinbarte Testfälle durchgespielt, mit denen dargelegt wird, dass die fertiggestellte Software die in der Anforderungsspezifikation gestellten Anforderungen erfüllt.

Zitierte und weiterführende Literatur

Berner, S., M. Arnold, S. Joos, M. Glinz (1996). *Java-Entwicklungsrichtlinien*. Institut für Informatik der Universität Zürich, Forschungsgruppe Requirements Engineering.

Berner, S., S. Joos, M. Glinz (1997). Entwicklungsrichtlinien für die Programmiersprache Java. *Informatik/Informatique* 4, 3 (Jun 1997). 8-11.

Frühauf, K., J. Ludewig, H. Sandmayr (1991). *Software-Prüfung. Eine Fibel*. Zürich: vdf und Stuttgart: Teubner.

Wirth, N. (1993). *Systematisches Programmieren: eine Einführung*. 6. Auflage. Stuttgart: Teubner.

9. Qualitätsmanagement

9.1 Grundlagen

Werden Produkte in kleinem, überschaubarem Rahmen entwickelt, hergestellt und vertrieben, so erübrigen sich spezielle Maßnahmen zur Sicherstellung der Qualität. Pfuscharbeit hat sofortige Rückwirkungen auf deren Verursacher, einerseits weil Hersteller und Abnehmer einander kennen, andererseits weil es im Kleinbetrieb eine wirksame Kontrolle und häufig ein traditionelles, qualitätsbewusstes Berufsethos gibt. Dies ist z.B. typisch der Fall in kleinen handwerklichen Meisterbetrieben. Qualitätsmanagement im eigentlichen Sinn gab es historisch erst mit der industriellen Massenproduktion, wo die Qualität in der Produktion vor allem durch statistische Verfahren sichergestellt wurde.

Qualitätsmanagement in der *Entwicklung* von Produkten ist nochmals wesentlich jünger, da Massenphänomene (Probleme, die sich auch in der Entwicklung nur noch durch Arbeitsteilung und Gruppenarbeit lösen lassen, zunehmende Anonymisierung, Entfremdung von Entwicklern und Produkten) dort erst in den vergangenen 30-40 Jahren aufgetreten sind.

Software-Qualitätsmanagement ist der jüngste Spross, bei dem es speziell um die Qualität von industriell entwickelter Software geht.

9.1.1 Definitionen

Im allgemeinen Sprachgebrauch wird Qualität meistens mit einem hohen Wert auf einer *absoluten* Güteskala assoziiert. Im Gegensatz dazu bezeichnet der industrielle Qualitätsbegriff eine *relative* Güte, die sich an der *Erfüllung zuvor aufgestellter Ziele* bemisst.

Definition 9.1. Qualität. Die Gesamtheit von Merkmalen einer Einheit bezüglich ihrer Eignung, festgelegte und vorausgesetzte Erfordernisse zu erfüllen (ISO 8402).

Eine *Einheit* kann ein Produkt, eine Dienstleistung, eine Tätigkeit, ein Prozess, ein System, eine Person, eine Organisation, etc. sein.

Qualität im Sinn von ISO 8402 ist folglich *Zielerfüllung*. Dennoch schwingt die Vorstellung hoher Güte immer mit, da eben auch „*vorausgesetzte* Erfordernisse“, das heißt solche, die nicht explizit als Ziele formuliert sind, zu erfüllen sind.

Die Gesamtheit der qualitätsrelevanten Tätigkeiten in einer Organisation wird als *Qualitätsmanagement* bezeichnet.

Definition 9.2. Qualitätsmanagement. Alle Tätigkeiten der Gesamtführungsaufgabe, welche die Qualitätspolitik, Ziele und Verantwortlichkeiten festlegen sowie diese durch Mittel wie Qualitätsplanung, Qualitätslenkung, Qualitätssicherung und Qualitätsverbesserung im Rahmen des Qualitätsmanagementsystems verwirklichen (ISO 8402).

Hinweis: Bis vor wenigen Jahren wurde das gesamte Qualitätsmanagement als „Qualitätssicherung“ bezeichnet. Seit 1994 wird in allen Normen die hier verwendete Terminologie benutzt. Der Begriff „Qualitätssicherung“ ist auf die Qualitätsmanagement-Darlegung, d.h. den Nachweis der Qualitätsmaßnahmen beschränkt worden. In der Praxis wird aber immer noch häufig von Qualitätssicherung gesprochen, wenn Qualitätsmanagement gemeint ist.

Qualitätsmanagement braucht einen organisatorischen Rahmen. Dieser wird durch das Qualitätsmanagementsystem gebildet.

Definition 9.3. *Qualitätsmanagementsystem.* Die Struktur, Verantwortlichkeiten und Mittel zur Verwirklichung des Qualitätsmanagements (ISO 8402).

9.1.2 Besonderheiten des Software-Qualitätsmanagements

Software nimmt insofern eine Sonderstellung ein, als

- Software ein Produkt ist, bei dem nur die Entwicklung schwierig ist, die Produktion dagegen aus bloßem, völlig unproblematischem Kopieren besteht
- es praktisch keine Traditionen für Software-Entwicklung gibt
- Software aufgrund ihres immateriellen Charakters schwierig zu überprüfen ist.

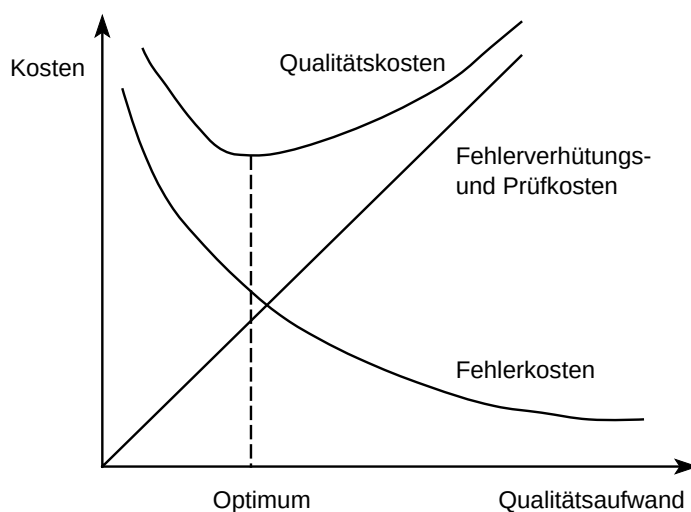
Bezogen auf Qualität heißt dies, dass die Qualität der Entwicklung, d.h. der *Produktschöpfung* gesichert werden muss, dass auf keine Traditionen und kein Berufsethos für die Herstellung qualitativ hochwertiger Software zurückgegriffen werden kann und dass die Qualitätsprüfung von Software schwierig ist. Vor allem das zweite sorgt für Schwierigkeiten bei der Einführung eines Qualitätsmanagementsystems und bei der notwendigen Verankerung des Qualitätsbewusstseins in den Köpfen der Beteiligten.

Grundsätzlich gibt es aber keinen wesentlichen Unterschied zwischen Software-Qualitätsmanagement und dem Qualitätsmanagement bei der Entwicklung anderer Produkte, auch wenn Software-Qualitätsmanagement verschiedentlich separat genormt worden ist (z.B. ANSI/IEEE-Normen).

9.1.3 Qualitätskosten

Qualität ist nicht umsonst. Einrichtung und Unterhalt eines Qualitätsmanagementsystems kosten erhebliche Summen. Die Durchführung von Qualitätsmanagement-Maßnahmen in den Projekten belastet diese sowohl terminlich wie kostenmäßig.

Welcher Nutzen steht diesen Kosten gegenüber? Qualitätsmanagement bedeutet Verhinderung bzw. rechtzeitige Entdeckung von Fehlern. Jeder nicht entdeckte oder zu spät entdeckte Fehler ist mit teilweise erheblichen Fehlerbeseitigungskosten und möglichen Folgekosten (z.B. Verlust von Marktanteilen) verbunden. Die Wirtschaftlichkeit des Qualitätsmanagements ergibt sich also aus der Gegenüberstellung von Qualitätsmanagementkosten einerseits und Fehlerkosten andererseits (Bild 9.1).



nach Frühauf, Ludewig, Sandmayr (1988)

Bild 9.1: Qualitätskosten

9.2 Das Qualitätsmanagementsystem

9.2.1 Das Qualitätswesen

Das Qualitätswesen ist eine Sekundärorganisation innerhalb eines Unternehmens, in der die Fachleute für Qualität zusammengefasst sind. An der Spitze der Qualitätsorganisation steht der Qualitätsleiter, welcher unmittelbar an die Geschäftsleitung berichtet (Bild 9.2).

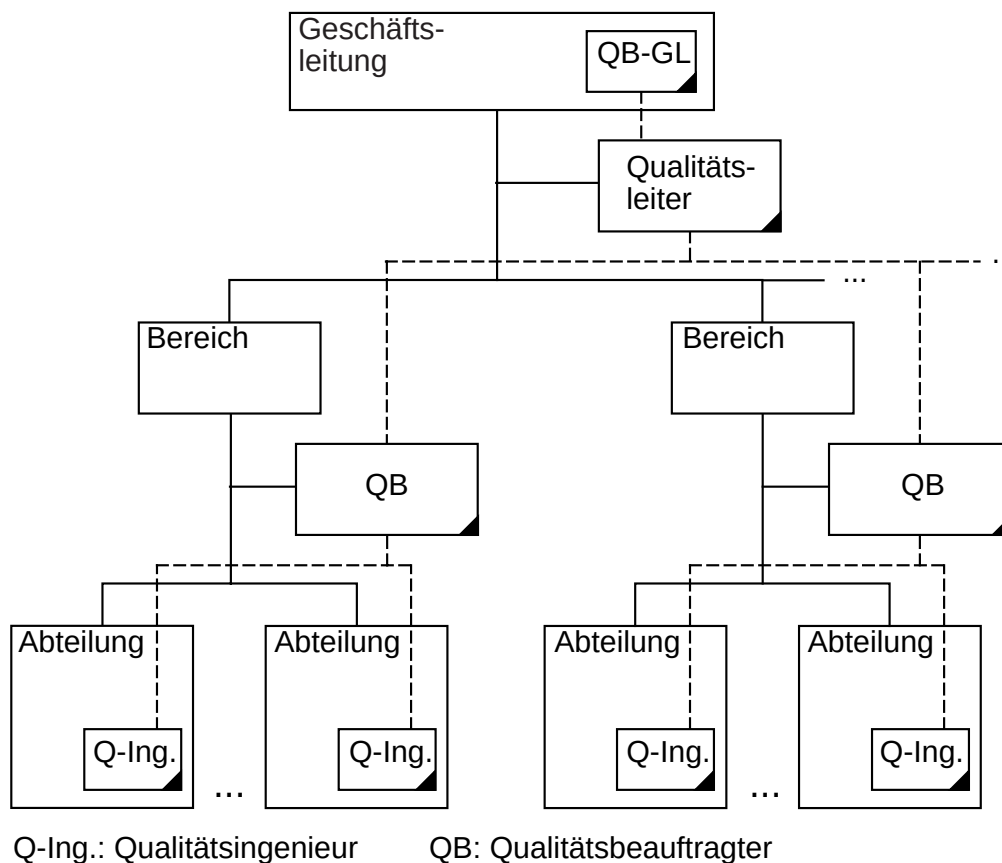


Bild 9.2: Das Qualitätswesen im Unternehmen

Im Qualitätswesen ist das *Fachwissen* über Qualität, insbesondere über Qualitätsmanagementverfahren, konzentriert. Die *Qualitätsverantwortung* liegt jedoch nicht allein beim Qualitätswesen, sondern muss vom ganzen Unternehmen getragen werden.

9.2.2 Grundsätze des Qualitätsmanagements

Im Qualitätsmanagement gelten die folgenden sechs Grundsätze.

Grundsatz 1. Qualität – in der Entwicklung wie in der Produktion – muss *erzeugt* werden, sie kann *nicht erprüft* werden.

Grundsatz 2. Qualität bezieht sich immer sowohl auf die hergestellten Produkte wie auf die Prozesse zur Herstellung (Entwicklung und Produktion) dieser Produkte.

Schlampige Herstellungsprozesse sind eine denkbar schlechte Voraussetzung für die Herstellung guter Produkte. Wer qualitativ hochwertige Produkte will (das, was letztlich zählt), muss daher auch für qualitativ einwandfreie Herstellungsprozesse sorgen.

Grundsatz 3. Die *Qualitätsverantwortung* liegt immer bei den gleichen Leuten, welche auch die Sach-, Termin- und Kostenverantwortung haben, d.h. bei den *Führungskräften* (in der Linie wie in Projektorganisationen) und bei den *Entwicklern*.

Grundsatz 4. Das *Qualitätswesen* ist verantwortlich für die *Ermittlung (Messung) der Qualität*. Es erbringt Dienstleistungen in allen Belangen der Qualität sowohl für die Entwickler wie für die Führungsverantwortlichen.

Das Qualitätswesen erarbeitet die Kriterien zur Qualitätsmessung, führt die Messungen durch oder ist dabei behilflich. Es erarbeitet für die Führungsverantwortlichen Empfehlungen zu anstehenden Entscheidungen (z.B. Verkaufsfreigabe eines Produkts). Es informiert die Entwickler über die Qualität ihrer Arbeit.

Grundsatz 5. Das Qualitätswesen muss einen unabhängigen Berichterstattungspfad haben, der bis zur Geschäftsleitung geht.

Die Führungsverantwortlichen können Empfehlungen des Qualitätswesens ignorieren, da sie gemäß Grundsatz 3 nicht nur die Termin-, Kosten- und Sachverantwortung, sondern auch die Qualitätsverantwortung haben. Da qualitativ schlechte Produkte jedoch den Ruf eines ganzen Unternehmens ruinieren können, muss das Qualitätswesen die Möglichkeit haben, auf einem ordentlichen Berichtspfad die Geschäftsleitung über Pfuscharbeit zu orientieren.

Grundsatz 6. Die Entwickler müssen über die Qualität ihrer Arbeit orientiert werden. Jeder Vorgesetzte muss die Qualität der Arbeit seiner Mitarbeiter in deren Beurteilung mit einbeziehen.

Entwickler sind nur dann motiviert, Qualitätsarbeit zu leisten, wenn sie wissen und spüren, dass Qualität belohnt und Pfusch bestraft wird.

Will man erreichen, dass Qualitätsmanagement etwas Aufbauendes, ein positiver Teil der Unternehmenskultur ist und nicht etwas Negatives, Destruktives, so ist sehr darauf zu achten, dass nicht nur Schlechtes kritisiert, sondern vor allem auch Gutes gelobt wird.

9.2.3 QM-Verfahren

Die Qualitätsmanagementverfahren kann man in drei Klassen gliedern: Qualitätsplanung, Qualitätslenkung und Qualitätsprüfung (Bild 9.3).

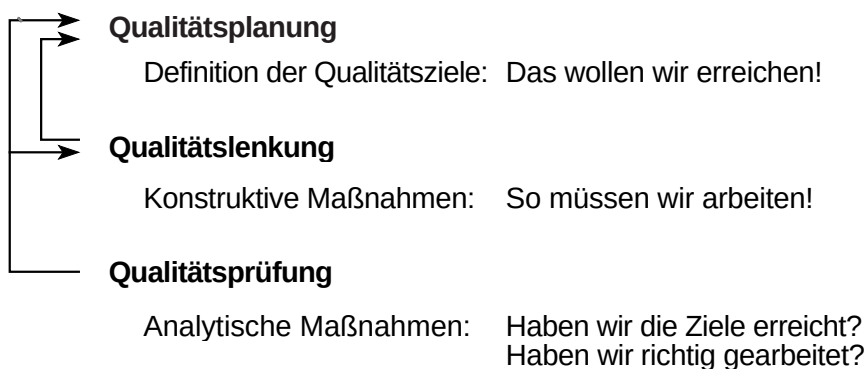


Bild 9.3: Qualitätsmanagement-Maßnahmen

Zur **Qualitätsplanung** im Allgemeinen gehört der Aufbau des Qualitätsmanagementsystems, seine Dokumentation im Qualitätshandbuch und seine Ausprägung im Qualitätswesen. Ferner

gehört dazu eine Festlegung von Qualitätsmerkmalen, die bei jeder Produktentwicklung gemessen werden sollen.

Qualitätsplanung im Speziellen ist die Festlegung der Ziele für ein Projekt bzw. Produkt (vgl. Kapitel 2).

Qualitätslenkung im Allgemeinen findet statt durch Festlegung von *Entwicklungsmethoden*, welche die Vorgehensweise definieren, durch *Sprachen*, welche die Ausdrucksweise normieren, und durch *Werkzeuge*, welche den Einsatz von Methoden und Sprachen unterstützen.

Zu den Methoden gehören auch Entwicklungsrichtlinien.

Ferner sind alle Maßnahmen zur Ausbildung und zur Vereinheitlichung der Arbeitsweise der Entwickler Qualitätslenkungs-Maßnahmen.

Qualitätslenkung im Speziellen sind alle Maßnahmen, die in der Führung eines Projekts ergriffen werden, um die Erreichung der gesetzten Ziele sicherzustellen.

Qualitätsprüfung wird in Abschnitt 9.3 besprochen.

9.2.4 Dokumentation des Qualitätsmanagementsystems

Das Qualitätsmanagementsystem wird wie folgt dokumentiert:

- Das *Qualitätshandbuch* enthält die Beschreibung des Qualitätsmanagementsystems eines Unternehmens oder Unternehmensbereichs.
Falls das Unternehmen Kunden hat, die über das Qualitätsmanagement orientiert werden wollen, bevor sie Aufträge erteilen, wird das Qualitätshandbuch in zwei Bände unterteilt. Der erste Band wird auf Anfrage an Kunden abgegeben. Er enthält die Qualitätsmanagement-Organisation und einen Überblick über die Qualitätsmanagement-Maßnahmen.
Der zweite Band ist das Handbuch der Qualitätsmanagement-Verfahren. Er beschreibt das unternehmensspezifische Qualitätsmanagement-Know-How und ist vertraulich.
- Der *Software-Qualitätsplan* enthält die Vorgaben betreffend Qualität für die Entwicklung von Produkten. In der Regel gibt es einen allgemeinen Rahmenplan, der bei Bedarf projektspezifisch ergänzt wird.
- Detaillierte Entwicklungsrichtlinien und Ausführungsbestimmungen, die den Rahmen eines Qualitätsplans sprengen würden, werden häufig in einem *Software-Entwicklungshandbuch* oder in *Software-Entwicklungsrichtlinien* (Berner et al. 1997) zusammengefasst.

9.2.5 Qualitätssicherung

Die Existenz eines Qualitätsmanagementsystems garantiert noch nicht dessen Wirksamkeit. Die *Darlegung des Qualitätsmanagements*, d.h. alle Tätigkeiten zur Schaffung von Vertrauen, dass die Qualitätsanforderungen erfüllt werden, wird Qualitätssicherung¹⁾ genannt. Als zentrale Maßnahmen braucht es regelmäßige Überprüfungen, ob das Qualitätsmanagementsystem wie geplant funktioniert und ob die vorgesehenen Qualitätsmaßnahmen wirklich durchgeführt werden. Solche Überprüfungen heißen *Audits*.

Interne Audits werden durch das Qualitätswesen geplant und durchgeführt. Jede Organisationseinheit sollte etwa einmal jährlich auditiert werden. Die Ergebnisse dieser Audits fließen in Maßnahmenpläne zur Verbesserung der Qualität und zur Eliminierung erkannter Schwachstellen ein.

¹⁾ Bis vor wenigen Jahren wurde das gesamte Qualitätsmanagement als „Qualitätssicherung“ bezeichnet. Seit 1994 wird in allen Normen die hier verwendete Terminologie benutzt. In der Praxis wird aber immer noch häufig von Qualitätssicherung gesprochen, wenn Qualitätsmanagement gemeint ist.

Bei *externen Audits* wird die Überprüfung durch eine externe, auf Qualitätsmanagement spezialisierte Organisation durchgeführt. Ein externer Audit ist unter anderem dann erforderlich, wenn eine Organisation ein *Zertifikat* für ihr Qualitätsmanagementsystem braucht oder will. Dies ist beispielsweise der Fall, wenn ein großer Kunde einen Auftrag davon abhängig macht, dass das Qualitätsmanagementsystem des Auftragnehmers gewissen Mindestanforderungen genügt oder wenn sich die Organisation von der Zertifizierung einen Wettbewerbsvorteil erhofft. In der Regel wird nach der Normenreihe ISO 9000 zertifiziert.

9.3 Qualitätsprüfung

9.3.1 Allgemeines

Die Qualitätsprüfung stellt fest, ob ein Arbeitsergebnis den Vorgaben entspricht. Letztlich ist zu prüfen, ob ein fertiggestelltes Produkt die an das Produkt gestellten Anforderungen erfüllt. Mit der Prüfung darf jedoch nicht bis zum Abschluss der Entwicklung gewartet werden, weil verfehlte Ziele dann möglicherweise nicht mehr, oder nur mit sehr hohen Kosten doch noch erreichbar sind.

Qualitätsprüfung ist daher eine permanente, die Entwicklung begleitende Aufgabe. Man unterscheidet zwei Arten der Überprüfung, die Validierung und die Verifikation (Bild 9.4).

Definition 9.4. *Validierung.* Der Prozess der Beurteilung eines Systems oder einer Komponente während oder am Ende des Entwicklungsprozesses, mit dem Ziel, festzustellen, ob die spezifizierten Anforderungen erfüllt sind. (IEEE 610.12).

Validierung ist folglich die Prüfung, ob ein Arbeitsergebnis geeignet ist, die Erwartungen der Endbenutzer zu erfüllen, d.h. ob überhaupt das *richtige Produkt* entwickelt wird.

Definition 9.5. *Verifikation.* 1. Der Prozess der Beurteilung eines Systems oder einer Komponente mit dem Ziel, festzustellen, ob die Resultate einer gegebenen Entwicklungsphase den Vorgaben für diese Phase entsprechen. 2. der formale Beweis der Korrektheit eines Programms. (IEEE 610.12).

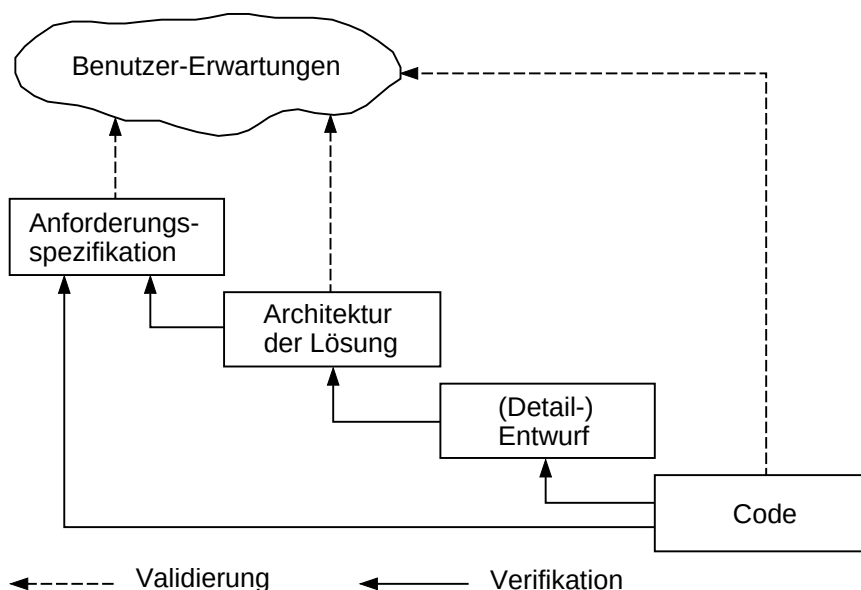


Bild 9.4: Validierung und Verifikation

Verifikation ist demnach Prüfung, ob ein Arbeitsergebnis den für den betreffenden Arbeitsschritt gegebenen Vorgaben genügt, d.h. ob das Produkt *richtig* entwickelt wird. Der Beweis der Korrektheit von Programmen ist ein Verifikationsverfahren auf der Grundlage formaler mathematischer Mittel.

Zur Validierung können grundsätzlich keine mathematisch formalen Verfahren verwendet werden, da die Benutzererwartungen nicht formalisierbar sind.

Bei der Verifikation der Zwischenprodukte einer Entwicklung dagegen ist es grundsätzlich möglich, mathematisch formale Verfahren zu verwenden, welche die Korrektheit der Entwicklung *beweisen*. Der dafür erforderliche Aufwand ist jedoch so groß, dass die formale Verifikation in der Praxis nur eine Nischenrolle spielt (zum Beispiel zur Entwicklung der hochkritischen Komponenten in sicherheitskritischen Systemen).

Die Prüfverfahren, die heute in der Praxis eingesetzt werden bzw. eingesetzt werden sollten, sind daher zwar *formalisiert* (d.h. folgen einem formellen Verfahrensprozess), aber nicht im mathematischen Sinn formal.

9.3.2 Prüfverfahren

Vier Prüfverfahren sind von praktischer Bedeutung: das Review, der Test, die Simulation und der Prototyp.

Ein *Review* ist eine formell organisierte Überprüfung eines Arbeitsergebnisses durch eine Gruppe von Gutachtern. Trotz nachgewiesener großer Wirksamkeit werden Reviews heute immer noch viel zu wenig eingesetzt. In Kapitel 9.4 ist daher die Reviewtechnik genauer erläutert.

Ein *Test* ist die Überprüfung eines Programms, ob dieses bei gegebenen Eingaben die erwarteten Resultate liefert. Nach dem heutigen Stand der Technik ist qualitativ hochwertige Software nur mit systematischem Testen (vgl. Kapitel 8.3) erstellbar.

In einer *Simulation* wird ein bestimmter Aspekt eines Systems modelliert und in seinem Verhalten überprüft. Simulationen dienen vor allem dazu, die Erreichung von Leistungsanforderungen zu überprüfen.

Ein *Prototyp* ist eine Vorab-Realisierung eines kritischen Systemteils. Er dient vor allem dazu, Anforderungen zu validieren und Konzepte auf Machbarkeit bzw. Einhaltung der Leistungsanforderungen zu überprüfen (vgl. Kapitel 3 und 4).

9.4 Reviewtechnik

9.4.1 Grundlagen

9.4.1.1 Definition, Ziele

Definition 9.6. *Review.* Eine formell organisierte Zusammenkunft von Personen zur inhaltlichen oder formellen Überprüfung eines Produktteils (Dokument, Programmstück, etc.) nach vorgegebenen Prüfkriterien und -listen.

Ein Review hat zum Ziel, sowohl die Schwachstellen und Mängel wie auch die Stärken des geprüften Produkts aufzuzeigen und damit dessen Qualität festzustellen. Man unterscheidet zwei Formen von Reviews: Inspektionen und Walkthroughs.

Eine *Inspektion* ist ein Review, bei dem der Prüfling von den Gutachtern systematisch Punkt für Punkt auf Stärken und Schwächen abgeklopft wird.

Ein *Walkthrough* ist ein Review, bei dem der Autor oder die Autorin die Funktionsweise des Prüflings Schritt für Schritt beschreibt, während die Gutachter aufmerksam zuhören und überall einhaken, wo sie Mängel entdecken.

In der Regel werden Reviews als Inspektionen durchgeführt. Dementsprechend gelten die Ausführungen in den folgenden Kapiteln primär für Inspektionen. Sie lassen sich sinngemäß aber auch auf Walkthroughs anwenden. Walkthroughs sind dann sinnvoll, wenn das Urteil von Gutachtern gefragt ist, welche das zu prüfende Produkt (oder die Sprache, in der es formuliert ist) nicht so gut kennen, dass sie es ohne Führung inspizieren können.

Neben dem zu prüfenden Material braucht es für ein Review sogenannte *Referenzunterlagen*. In einem Code-Review kann beispielsweise die inhaltliche Richtigkeit von Algorithmen und Datenstrukturen nur geprüft werden, wenn als Referenz ein Konzept- oder Entwurfsdokument vorhanden ist, in welchem entsprechende Vorgaben stehen.

Unter Referenzunterlagen versteht man einerseits Unterlagen, welche sachlich-inhaltliche Vorgaben enthalten, die der Prüfling erfüllen muss und andererseits Standards und Prüflisten, anhand derer die Prüfung durchgeführt werden soll.

Reviews können in jedem Stadium der Software-Entwicklung eingesetzt werden. In den frühen Phasen der Entwicklung ist das Review das einzige Mittel für eine fundierte Überprüfung von Resultaten. In der Codierung und Integration werden Reviews als zusätzliches Prüfmittel neben den Tests eingesetzt.

9.4.1.2 Motivation

Sorgfältig durchgeführte Reviews verbessern die Qualität der erzeugten Software erheblich. Fagan (1986) berichtet von zwei Projekten, bei denen in den Code-Reviews 82% bzw. 93% aller Fehler entdeckt wurden, und dies gemessen über die gesamte Lebensdauer der jeweiligen Software! Andere Quellen sprechen von Senkungen der Restfehlerraten (Fehler pro 1000 Zeilen ausgeliefertem Code) um bis zu zwei Drittel.

Solche Qualitätsverbesserungen schlagen nicht nur in höherer Kundenzufriedenheit zu Buche, sondern reduzieren sowohl die Entwicklungskosten (weniger Aufwand für Test und Fehlerbehebung) als auch die Pflegekosten.

9.4.1.3 Positive Nebeneffekte

Werden in einer Organisation regelmäßig Reviews durchgeführt, so hat dies einen ständigen *Wissens- und Kenntnistransfer* von den besseren zu den schlechteren Leuten zur Folge. Jeder Review-Teilnehmer lernt aus den Stärken und Schwächen in den Produkten anderer etwas für die eigene Arbeit. Auf die gleiche Weise wird eine einheitliche Anwendung von Standards und Konventionen innerhalb der Organisation gefördert.

Die in Reviews festgestellten Mängel lassen sich statistisch auswerten. Die Ergebnisse können zur *Qualitätslenkung* verwendet werden. Tritt beispielsweise eine bestimmte Art von Fehlern besonders häufig auf, so können gezielt Maßnahmen ergriffen werden, damit diese Fehler nicht mehr gemacht werden.

9.4.2 Die Durchführung von Reviews

Effektivität und Erfolg von Reviews hängen wesentlich von der richtigen Durchführung ab. Dazu gehört nicht nur die Abwicklung der eigentlichen Review-Sitzung nach besonderen Regeln, sondern ein insgesamt wohlorganisiertes Umfeld.

9.4.2.1 Der Ablauf eines formellen Review

Zu einem formellen Review gehören die nachfolgend beschriebenen fünf Schritte.

1. Planung

Reviews müssen in der Entwicklung eingeplant werden. Der notwendige Aufwand an Zeit und Arbeitskapazität ist zu budgetieren. Die Review-Teilnehmer müssen rechtzeitig eingeladen und mit dem zu prüfenden Material versorgt werden. Je nach Qualitätsanforderungen können die Aufwendungen für Reviews bis zu 15% des gesamten Entwicklungsaufwandes ausmachen.

2. Vorbereitung

Die Teilnehmer an einem Review erhalten das zu prüfende Material sowie die Referenzunterlagen im Voraus und bereiten sich individuell auf die Sitzung vor. Sie arbeiten das Material durch und markieren bzw. notieren alle ihre Feststellungen über Schwächen und Stärken. Die eigentliche Review-Sitzung dient nicht zum *Suchen*, sondern zum *Zusammentragen und Bewerten* von Schwächen und Stärken. Review-Sitzungen mit unvorbereiteten Teilnehmern sind Zeitverschwendung. Vorgegebene Prüflisten und Standards erleichtern die Arbeit.

3. Sitzung

Die Review-Sitzung wird von einem *Moderator* geleitet. Dieser sorgt für einen geordneten Sitzungsablauf und wacht über die Einhaltung der Review-Regeln. Zu den Aufgaben des Moderators gehören außerdem die Einladung zur Sitzung, die Überwachung der rechtzeitigen Verteilung des zu prüfenden Materials sowie die Weiterleitung des Review-Berichts an die verantwortliche Stelle.

4. Review-Bericht

Während der Sitzung wird ein Review-Bericht erstellt, welcher die in der Sitzung genannten und von der Mehrheit der Teilnehmer gebilligten Schwachstellen und Stärken auflistet. Der Review-Bericht soll einen Konsens der am Review Beteiligten ausdrücken; er wird am Ende der Sitzung von allen unterschrieben.

5. Überarbeitung und Nachkontrolle

Der Projektverantwortliche entscheidet aufgrund des Review-Berichts über die durchzuführenden Änderungen und gibt diese in Auftrag. Je nach Umfang dieser Änderungen erfolgt die Nachkontrolle durch ein neues Review oder durch den Projektverantwortlichen.

9.4.2.2 Informelle Reviews

Der mit formellen Reviews verbundene Organisationsaufwand scheint auf den ersten Blick beträchtlich. Man fragt sich, ob spontane, informelle Zusammenkünfte zum Finden, Besprechen oder Lösen von Problemen nicht viel einfacher und nützlicher wären. Die Erfahrung zeigt, dass dies nicht zutrifft. Da informelle Reviews nicht geplant werden, sind sie keine systematische qualitätssichernde Maßnahme, sondern es bleibt weitgehend dem Zufall überlassen, ob ein Produktteil ein Review durchläuft oder nicht. Man hat außerdem festgestellt, dass die Fehleraufdeckungsrate in informellen Reviews deutlich niedriger ist als in formellen Reviews.

9.4.3 Rollen und Aufgaben der am Review Beteiligten

9.4.3.1 Moderator

Der Moderator organisiert und leitet die Review-Sitzung (vgl. 9.4.2.1, Punkt 3). Er sorgt ferner für die Fertigstellung und Weiterleitung des Review-Berichts.

Im Idealfall ist der Moderator ein Mitarbeiter des Qualitätswesens und hat ein spezielles Training für die Leitung von Reviews absolviert.

Die Effektivität eines Reviews hängt stark vom Geschick des Moderators ab. Es geht unter anderem darum

- eine offene Atmosphäre zu schaffen, in der jeder sich frei äußern kann, ohne Angst sich zu blamieren
- Zurückhaltende Teilnehmer zu Aussagen zu ermuntern
- Dauerredner zu bremsen
- Streit und persönliche Angriffe zu vermeiden bzw. beizulegen
- Abgleiten in nutzlose Detaildiskussionen zu verhindern
- immer wieder Konsens herzustellen
- generell über die Einhaltung der Review-Regeln zu wachen und diese freundlich, aber bestimmt durchzusetzen.

9.4.3.2 Schreiber

Der Schreiber erstellt den Review-Bericht. Dieser entsteht für alle sichtbar, indem der Schreiber während der Sitzung fortlaufend am Hellraumprojektor auf Folien schreibt. Die übrigen Teilnehmer können so das Geschriebene laufend verfolgen und Einspruch erheben, wenn sie meinen, es sei etwas nicht adäquat notiert worden. Schreiber und Moderator sollten möglichst nicht die gleiche Person sein.

9.4.3.3 Gutachter

Die übrigen Review-Teilnehmer haben die Aufgabe, das zu prüfende Material in der Vorbereitung zu begutachten und in der Sitzung über ihre Erkenntnisse zu berichten. Die Autoren des zu prüfenden Materials können nicht gleichzeitig Gutachter sein. Die Gutachter sind in der Regel Entwickler aus anderen Projekten der gleichen Abteilung oder des gleichen Bereichs. Gegebenenfalls können für bestimmte Probleme Fachgutachter oder Qualitätsingenieure herangezogen werden. Die Gutachter sollen

- sich sorgfältig vorbereiten
- alle wichtigen Punkte in der Sitzung nennen
- sich nicht persönlich profilieren wollen und nicht alles besser wissen
- den anderen gut zuhören und nicht bereits genannte Dinge erneut aufbringen
- während der Sitzung zu einer positiven, offenen und kooperativen Atmosphäre beitragen.

Der Schreiber kann gleichzeitig auch Gutachter sein.

9.4.3.4 Autor

Der Autor (bzw. ein Vertreter der Autorenschaft) nimmt aus zwei Gründen an der Sitzung teil: Erstens können Unklarheiten und Missverständnisse unter Umständen sofort geklärt und mit einer entsprechenden Notiz im Review-Bericht erledigt werden. Zweitens müssen die Autoren später anhand des Review-Berichts die festgestellten Mängel beheben. Die richtige Interpretation

der Befunde im Review-Bericht fällt wesentlich leichter, wenn ein Vertreter der Autorenschaft bei der Abfassung dieses Berichts dabei gewesen ist.

9.4.3.5 Die Qualitätsverantwortung

Die Review-Teilnehmer sind verantwortlich für die *Qualität des Reviews*, d.h. dafür, dass einerseits schlechte Produkte als schlecht erkannt werden und möglichst alle Mängel im Review-Bericht aufgeführt sind, und dass andererseits gute Produkte als gut erkannt und nicht mit gesuchten und nebensächlichen Schwachstellen belastet werden.

Sie sind jedoch *nicht* verantwortlich für die Qualität des *Produkts*. Weder können sie für festgestellte Schwächen haftbar gemacht werden, noch dafür, dass festgestellte Mängel unter Umständen nicht behoben werden.

Die Verantwortung für die Qualität eines Produkts muss bei der selben Person liegen wie die Verantwortung für Sachziele, Kosten und Termine. Diese Person ist verantwortlich sowohl für die Veranlassung der Reviews als auch für die Veranlassung und Nachkontrolle der notwendigen Verbesserungen.

9.4.4 Review-Regeln

- Das zu prüfende Material wird rechtzeitig vor der Sitzung verteilt. Ein „Review“ von Tischvorlagen ist Zeitvergeudung.
- Es muss dafür gesorgt sein, dass die Teilnehmer rechtzeitig eingeladen werden und zu diesem Termin auch frei sind. Sollen Reviews zum festen Bestandteil einer Entwicklungsmethode werden, so empfiehlt es sich, einen Zeitblock auf einen festen Wochentag zu legen (z.B. immer Dienstags 10-12h), in dem Reviews durchgeführt werden und auf den nur in Notfällen andere Termine gelegt werden dürfen.
- Alle kommen vorbereitet zur Sitzung. Der Moderator bricht das Review ab, wenn sich zeigt, dass mehrere Teilnehmer nicht vorbereitet sind.
- An einem Review nehmen mindestens drei und höchstens etwa sieben Personen teil. Bei zu wenig Personen geht der zentrale Effekt des Gruppen-Gutachtens verloren. Ab etwa sieben teilnehmenden Personen tragen weitere Teilnehmer nur noch marginal zur Liste der Befunde bei, folglich wird das Review mit zunehmender Teilnehmerzahl immer unwirtschaftlicher.
- Eine Sitzung dauert maximal 2 Stunden. Bei längeren Sitzungen lässt die Konzentration nach und die Effektivität des Reviews sinkt rapide. Dementsprechend muss der Umfang des zu prüfenden Materials gewählt werden. Richtwerte: für Code-Reviews ca. 150-200 Codezeilen (ohne Kommentare), für Dokument-Reviews ca. 25 Seiten pro Sitzung.
- Die Probleme werden in der Sitzung nur genannt, nicht gelöst. Problemlösungsdiskussionen werden vom Moderator möglichst im Keim erstickt. Eine informelle Nachsitzung (z.B. bei einem gemeinsamen anschließenden Mittagessen) kann solchen Problemlösungsgesprächen Raum bieten.
- Jeder Gutachter nennt mindestens einen positiven und einen negativen Punkt im zu prüfenden Material.
- Stilfragen werden nicht diskutiert. Dort, wo vorgeschriebene Standards auch Stilfragen regeln, werden Abweichungen vom Standard als Mängel genannt, in allen anderen Fällen ist der Stil Ermessenssache der Autoren.
- Das Produkt wird bewertet, nicht deren Produzenten. Reviews sind dazu da, die Qualität von Produkten zu beurteilen. Sie sollen und dürfen nicht dazu verwendet werden, die Fähigkeiten der Personen, welche das Produkt erzeugt haben, zu beurteilen. Dies gilt sowohl für die Review-Teilnehmer wie für die Manager. Die Teilnehmer müssen sich mit persönlichen Beurteilungen zurückhalten, um die positive Atmosphäre während des Reviews nicht zu ge-

fährden und um den Autor, welcher beim Review anwesend ist, nicht über das unvermeidliche Maß hinaus psychisch zu belasten. Die Manager dürfen die Review-Ergebnisse nicht zur Mitarbeiterbeurteilung heranziehen, wenn sie nicht wollen, dass die Reviews massiv an Effektivität verlieren, weil die Gutachter ihre Kollegen, welche das geprüfte Produkt erzeugt haben, schonen.

- Die Prüfung ist umso einfacher und effizienter, je mehr Standards und Prüflisten bereitstehen, nach denen bewertet werden kann. Wo keine solchen vorhanden sind, sollten sie daher unbedingt geschaffen bzw. eingeführt werden.
- Häufig werden beim Review nicht nur Befunde im Prüfling, sondern auch Fehler in den Referenzunterlagen gefunden. Solche Befunde werden ebenfalls erhoben und auf einer separaten Befundliste notiert.

9.4.5 Der Review-Bericht

Für den Review-Bericht werden mit Vorteil vorbereitete Formblätter verwendet. Ein Beispiel, wie solche Formblätter aussehen können, findet sich im Anhang. Das Deckblatt wird vom Moderator vorbereitet. Am Ende der Sitzung wird dort der Gesamtbefund notiert und von allen Beteiligten mit ihrer Unterschrift bestätigt. Daran angeheftet wird die Liste der Befunde. Dies sind Papierkopien von allen Folien, die der Schreiber während der Sitzung beschrieben hat.

9.5 Testen

Das Thema Testen wird hier nur kurz behandelt, weil dieses Skript sich auch an Leute richtet, die nicht Informatiker sind, aber mit Software umgehen müssen. Im Übrigen wird auf die einschlägige Literatur verwiesen.

9.5.1 Grundsätze

Mit Tests werden zwei Ziele verfolgt:

- Im Test sollen möglichst viele der in einem Programm vorhandenen Fehler gefunden werden (Myers, 1979: *Testen* ist der Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden.)
- Aus der Tatsache, dass das Programm mit einer Menge von Testdaten fehlerfrei läuft, soll mit brauchbarer statistischer Sicherheit darauf geschlossen werden, dass das Programm auch im Einsatz fehlerfrei läuft.

Die *Korrektheit* eines Programms kann durch Testen (außer in trivialen Fällen) *nicht bewiesen* werden. Der Grund dafür ist einfach. Für einen Korrektheitsbeweis müssten alle Kombinationen aller möglichen Werte der Eingabedaten getestet werden. Schon für ein triviales Programm, das lediglich zwei mit 16 Bit repräsentierte ganze Zahlen addiert, wären $2^{16} \times 2^{16}$ Fälle zu testen.

Testen setzt voraus, dass die erwarteten Ergebnisse bekannt sind. Folglich muss entweder gegen eine *Spezifikation* oder gegen *vorhandene Testergebnisse* getestet werden. Letzteres ist zum Beispiel der Fall, wenn Tests nach Programmmodifikationen wiederholt werden (sogenannter *Regressionstest*). Unvorbereitete und undokumentierte Tests sind sinnlos.

Nicht alle Eigenschaften eines Programms können mit Testen geprüft werden. Qualitätsmerkmale wie zum Beispiel Portabilität oder Pflegbarkeit sind nicht testbar.

Mit Testen werden nur *Fehlersymptome*, nicht aber die *Fehlerursachen* gefunden. Auf jeden Test muss daher eine Phase folgen, in der die Ursachen zu den festgestellten Symptomen gefunden und behoben werden.

9.5.2 Testablauf

In jedem Test gibt es drei Phasen: Die Planung, die Vorbereitung und die Durchführung des Tests.

In der *Testplanung* wird die Teststrategie festgelegt und werden im Rahmen der Projektplanung die notwendigen Arbeiten und die dafür erforderlichen Ressourcen geplant.

In der *Testvorbereitung* werden die zu testenden Testfälle ausgewählt, eine Testumgebung zur Abarbeitung dieser Testfälle bereitgestellt und die Testvorschriften (d.h. die Vorgabedokumente, gemäß denen die Tests durchgeführt werden) erstellt.

Bei der *Testdurchführung* wird zunächst die Testumgebung nach den Angaben der für diesen Test geltenden Testvorschrift eingerichtet. Dann werden die Testfälle der Testvorschrift der Reihe nach ausgeführt. Alle Befunde werden notiert. Während der Testdurchführung wird der Prüfling nicht verändert. Die Testergebnisse werden in einer Testzusammenfassung festgehalten.

Die *Fehlerbehebung* ist kein Bestandteil des Tests. Sie erfolgt nachher, indem die Testbefunde analysiert, die Fehlerursachen bestimmt und die Fehler dann behoben werden.

Es ist sehr wichtig, dass die Aufwendungen für das Testen in jedem Projekt sorgfältig geschätzt und in der Projektplanung berücksichtigt werden. Nur dann ist es möglich, die für das Erreichen der angestrebten Qualität notwendigen Tests auch tatsächlich durchzuführen.

9.2.3 Testverfahren

Man unterscheidet zwei Klassen von Testverfahren. Beim *funktionsorientierten* Test (Black-Box-Test) erfolgt die Auswahl der Testfälle aufgrund der Spezifikation. Die Programmstruktur kann unbekannt sein. Es wird angestrebt, die geforderten Eigenschaften des Prüflings (Funktionalität, Leistungsverhalten, weitere Attribute) des Prüflings möglichst vollständig durch Testfälle abzudecken.

Beim *strukturorientierten* Test (White-Box-Test, Glass-Box-Test) erfolgt die Auswahl der Testfälle aufgrund der Programmstruktur. Es wird angestrebt, die Programmstruktur möglichst vollständig abzudecken (zum Beispiel alle Anweisungen mindestens einmal zu durchlaufen). Die Spezifikation muss ebenfalls bekannt sein, damit festgelegt werden kann, welche Resultate mit den ausgewählten Testdaten zu erwarten sind.

Für Einzelheiten wird auf die Literatur (zum Beispiel Beizer 1995, Frühauf, Ludewig, und Sandmayr 1991, Liggesmeyer 1990, Myers 1979) verwiesen.

Aufgaben

- 9.1 Sie arbeiten an der Entwicklung eines Software-Produkts mit. Wer ist für die Qualität dieses Produkts verantwortlich?
- 9.2 Sie sind Projektleiter in einem Software-Entwicklungsprojekt. Soeben haben Sie den Review-Bericht über das Lösungskonzept der zu entwickelnden Software erhalten. Der Bericht weist neben einer Reihe leichter Fehler drei schwere Fehler und eine als kritisch

eingestufte Auslassung aus. Sie sind gegenüber dem Zeitplan drei Wochen in Verzug und möchten daher so rasch wie möglich mit dem Detailentwurf und der Codierung beginnen lassen. Wie verhalten Sie sich? Was halten Sie von folgenden Maßnahmen?

- a) Sie lassen alle festgestellten Mängel beheben und beginnen nicht mit dem Detailentwurf, bis das Dokument das Nachreview bestanden hat und freigegeben ist.
- b) Sie lassen sofort mit Detailentwurf und Codierung beginnen und beschließen, die Fehler und Lücken erst zu beseitigen, wenn man beim Entwerfen bzw. Codieren an diese Stellen kommt.
- c) Sie gehen den Review-Bericht durch, markieren die Punkte, die Ihnen schwerwiegend erscheinen und lassen nur diese beheben.
- d) Sie veranlassen die Behebung der festgestellten Fehler, lassen aber gleichzeitig mit dem Detailentwurf eines Teilsystems beginnen, in dessen Konzept nur zwei leichte Mängel gefunden wurden.

9.3 Inspizieren Sie die im Anhang 2 gegebene Prozedur. Bereiten Sie sich vor, indem Sie die Prozedur durcharbeiten und Ihre Befunde notieren. Tun Sie sich mit drei oder vier Kolleginnen und Kollegen zusammen und führen Sie eine Review-Sitzung durch. Vergleichen Sie die Befundliste der Sitzung mit Ihren eigenen Befunden.

9.4 Warum sind unvorbereitete und undokumentierte Tests sinnlos?

Zitierte und ergänzende Literatur

Beizer, B. (1995). *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. New York etc.: John Wiley & Sons.

Berner, S., S. Joos, M. Glinz (1997). Entwicklungsrichtlinien für die Programmiersprache Java. *Informatik/Informatique* **4**, 3 (Jun 1997). 8-11.

Dunn, R. (1990). *Software Quality - Concepts and Plans*. Englewood Cliffs, N.J.: Prentice-Hall. [in dt. Übersetzung: *Software-Qualität. Konzepte und Pläne*. München: Hanser, 1993]

Fagan, M.E. (1986). Advances in Software Inspections. *IEEE Transactions on Software Engineering*, **SE-12**, 7(July 1986). 744-751.

Freedman D.P., G.M. Weinberg (1982). *Handbook of Walkthroughs, Inspections and Technical Reviews*. 3rd edition. Boston, Toronto: Little, Brown and Co.

Frühauf, K., J. Ludewig, H. Sandmayr (1991). *Software-Projektmanagement und -Qualitätssicherung*. 2. Auflage. Zürich: vdf und Stuttgart: Teubner.

Frühauf, K., J. Ludewig, H. Sandmayr (1991). *Software-Prüfung. Eine Fibel*. Zürich: vdf und Stuttgart: Teubner.

Liggesmeyer, P. (1990). *Modultest und Modulverifikation*. Reihe Angewandte Informatik Bd. 4. Mannheim, etc.: BI-Wissenschaftsverlag.

Liggesmeyer P., H.M. Sneed, A. Spillner (Hrsg.) (1992). *Testen, Analysieren und Verifizieren von Software*. Berlin, etc.: Springer.

Myers, G.J. (1979). *The Art of Software Testing*. New York: John Wiley & Sons. [in dt. Übersetzung: *Methodisches Testen von Programmen*. 4. Auflage. München: Oldenbourg, 1991.]

Russell, G.W. (1991). Experience with Inspection in Ultralarge-Scale Developments. *IEEE Software* **8**, 1 (Jan. 1991). 25-31.

Schnurer, K.E. (1988) Programminspektionen. *Informatik-Spektrum* **11**, 6 (Dez. 1988). 312-322.

Normen

EN 29000 = ISO 9000 (1987). Qualitätsmanagement- und Qualitätssicherungsnormen; Leitfaden zur Auswahl und Anwendung

EN 29001 = ISO 9001 (1987). Qualitätssicherungssysteme; Modell zur Darlegung der Qualitätssicherung in Design/Entwicklung, Produktion, Montage und Kundendienst

EN 29004 = ISO 9004 (1987). Qualitätsmanagement und Elemente eines Qualitätssicherungssystems; Leitfaden

DIN ISO 9000 Teil 3 (1990). Qualitätsmanagement- und Qualitätssicherungsnormen; Leitfaden für die Anwendung von ISO 9001 auf die Entwicklung, Lieferung und Wartung von Software

IEEE (1986). Standard for Software Verification and Validation. ANSI/IEEE Std 1012-1986. IEEE Computer Society Press.

IEEE (1989a). IEEE Standard Dictionary of Measures to Produce Reliable Software. IEEE Std 982.1-1988. IEEE Computer Society Press.

IEEE (1989b). IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software. IEEE Std 982.2-1988. IEEE Computer Society Press.

IEEE (1990). Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990. IEEE Computer Society Press.

ISO 8402 (1994). Qualitätsmanagement und Qualitätssicherung. Begriffe.

ISO 8402 (1994). Quality Management and Quality Assurance: Vocabulary.

ISO 9001 (1994). Quality Systems – Model for Quality Assurance in Design, Development, Production, Installation and Servicing. Second Edition. [Technisch revidierte Neuausgabe von ISO 9001:1987]

Anhänge:

Anhang 1: Formblätter für Review-Bericht:

- Deckblatt mit Review-Zusammenfassung
- Liste der Befunde

Anhang 2: Unterlagen für die Codeinspektion aus Aufgabe 9.3

Review-Bericht	Review-Nr.: _____		Seite _____ von _____	
	Datum: _____		Zeit von: _____ bis: _____	
Zu prüfendes Produkt				
Nummer / Version		Titel		
Referenzunterlagen				
Nummer / Version		Titel		
Bewertung	☐ akzeptiert (kein neues Review erforderlich) ☐ wie es ist ☐ nicht akzeptiert (neues Review erforderlich) ☐ kleine Änderungen ☐ Review nicht beendet ☐ große Änderungen ☐ Überarbeitung			
Review-Team				
Name	Funktion	Vorbereitungszeit	Unterschrift	
Freigabe (nach Nachkontrolle)				
Name	Datum	Unterschrift		

Anhang 2: Unterlagen für die Codeinspektion aus Aufgabe 9.3

1. Zu inspizierendes Codestück (in Modula-2)

```

PROCEDURE KennlinieTabellieren (b:REAL; e:REAL; d: REAL);
(* Tabelliert die verlangte Funktion im Bereich von b bis e *)
VAR lfd:      REAL; (* laufender x-Wert für Fkt.-Berechnung *)
    FktWert: REAL; (* berechneter Funktionswert *)

BEGIN
  (* Titel schreiben, Werte initialisieren: *)
  WriteString ("      x  F(x)=2x**2 e**(x**2) -3");
  WriteLn;
  lfd := b;
  (*Tabellierschleife: *)
  REPEAT
    FktWert := 2.0*3.141*lfd*lfd*exp(lfd*lfd) - 3.0;
    WriteReal (lfd, 10); WriteReal (FktWert, 12); WriteLn;
    lfd := lfd + d; (* laufenden Wert um Schrittweite erhöhen *)
  UNTIL lfd = e;
END KennlinieTabellieren;

```

Hinweise zum Lesen von Programmen in Modula-2

- `WriteString` schreibt eine Zeichenkette auf ein Ausgabemedium.
- `WriteReal (zahl, n)` konvertiert `zahl` in eine Zeichenkette und schreibt diese auf einer Breite von insgesamt `n` Zeichen auf ein Ausgabemedium.
- Die von mehreren `Write`-Anweisungen erzeugten Ausgaben schließen unmittelbar und ohne Zwischenraum aneinander an.

Durch den Aufruf von `WriteLn` wird eine neue Zeile auf dem Ausgabemedium begonnen.

- Es gibt keinen Operator für Exponentiation. Die Berechnung von x^2 wird als `x*x` programmiert.
- `exp (x)` ist eine eingebaute Standardfunktion, welche die Exponentialfunktion e^x berechnet.

2. Auszug aus dem Entwurfsdokument:

...

Prozedur Kennlinie tabellieren:

Die Kennlinie, welche durch die Funktion

$$f(x) = 2\pi x^2 e^{(x^2)} - 3$$

gegeben ist, wird im Bereich (x_1, x_2) mit der Schrittweite Δx tabelliert. Für den Fall $x_2 < x_1$ soll die Tabellierung unterbleiben. Über der Tabelle wird ein Titel

x f(x)=...
ausgedruckt.

...

3. Auszug aus den Codier-Richtlinien:

-
-
- Jede Anweisung beginnt auf einer neuen Zeile.
- Die Körper aller Prozeduren und Blöcke sind jeweils 2 Zeichen einzurücken.
- Namen sind so zu wählen, dass sie möglichst selbstdokumentierend sind.
- Namen, die aus drei und weniger Zeichen bestehen, sollen möglichst nur für lokale Aufgaben (Schleifenindizes o.ä. verwendet werden. Ihre Bedeutung ist bei der Definition zu dokumentieren.
- Konstante Größen sind am Beginn der Programme bzw. Prozeduren als symbolische Konstanten mit einem selbsterklärenden Namen zu vereinbaren.
- Jede Prozedur hat einen Kommentarkopf, in dem die Aufgabe der Prozedur sowie die Bedeutung der Prozedurparameter erklärt sind.
-
-

4. Auszug aus der Prüfliste für Codeinspektionen

-
-
- Übereinstimmung des Codes mit Entwurfsdokument:
 - Schnittstelle?
 - Datenstrukturen adäquat?
 - Algorithmen richtig? effizient?
 - geforderte Ein-/Ausgaben vorhanden?
- alle verwendeten Variablen initialisiert?
- Schleifen:
 - Verhalten am Anfang?
 - Verhalten am Ende; terminiert die Schleife sicher?
 - Weist die Schleife illegale Durchläufe ab?
 - Hat es schleifeninvarianten Code in der Schleife?
 - falls Laufvariablen vorhanden:
 - Laufvariable initialisiert?
 - Fortschaltung der Laufvariablen vorhanden und richtig?
- Ausgabe:
 - Zeilenende mit Zeilenfortschaltbefehl abgeschlossen?
 - tabellarisch aufgebaute Ausgaben sauber untereinander?
-
-

10. Dokumentation

10.1 Grundsätze der Dokumentation

Bei der Entwicklung von Software entsteht kein materielles Produkt. Alle Entwicklungstätigkeiten manifestieren sich letztlich nur in Form von Dokumenten (im weitesten Sinn; auch Programmcode ist ein Dokument). Den Dokumenten kommt daher in der Software-Entwicklung eine besondere Bedeutung zu.

10.1.1. Dokumentationsarten

Es gibt zwei Arten von Dokumentation bei der Software-Erstellung.

- Die *Produktdokumentation* dokumentiert ein Software-Produkt und seine Benutzung.
- Die *Projektdokumentation* dokumentiert die Entwicklung eines Software-Produkts.

Die Produktdokumentation wird manchmal auch Systemdokumentation genannt.

10.1.2 Aufgaben der Dokumentation

Dokumentation hat die folgenden drei Hauptaufgaben.

Wissenssicherung: Die Information aus den Köpfen holen

Das Wissen (Know-How) über ein System macht einen beträchtlichen Teil des Werts eines Systems aus. Die Produktdokumentation hat die Aufgabe, dieses Wissen schriftlich oder auf Datenträgern festzuhalten, damit es nicht verloren geht. Die Projektdokumentation sichert die Erfahrungen in der Abwicklung von Projekten.

Kommunikation: Reden miteinander genügt nicht

Eine geordnete Systementwicklung und -pflege ist ohne Kommunikation nicht möglich. Dies gilt sogar für Ein-Personen-Projekte, wenn das Produkt länger als ein paar Wochen leben soll. Mündliche Kommunikation ist bei Arbeiten in einem kleinen Personenkreis sehr effizient. Ausschließlich mündliche Kommunikation verursacht jedoch erhöhte Kosten (und wird letztlich ineffizient), wenn der Personenkreis sich ändert oder die Systembetreuung auf einen anderen Personenkreis übergeht. Letzteres ist typisch der Fall beim Übergang von der Entwicklung zur Nutzung.

Daher müssen auch in Kleinprojekten alle für ein System wichtigen mündlichen Informationen und Absprachen in Dokumenten festgehalten werden. Mündliche Kommunikation genügt nicht, da Erzeugung und Benutzung von Informationen unter Umständen um Jahre auseinander liegen können.

Sichtbarmachen des Projektfortschritts: Dokumente sind greifbare Resultate

Der Abschluss jeder Phase der Entwicklung wird nachprüfbar(!) markiert durch die Fertigstellung und Freigabe von Dokumenten. Bei den Produktdokumenten wird stets die Endfassung eines Dokuments erzeugt (welche nur noch über ein geordnetes Änderungswesen modifizierbar ist). Das Erstellen von Zwischendokumenten nur mit dem Zweck des Nachweises abgeschlossener Tätigkeiten ist zu vermeiden.

Dadurch, dass die Abschlussdokumentation schritthaltend mit der Entwicklung entsteht, ist der Projektfortschritt besser quantifizierbar.

10.1.3 Wirtschaftlichkeit der Dokumentation

Dokumentation kostet Entwicklungszeit und -geld, darum wird sie – vor allem unter Termin- druck – oft nicht oder nur fragmentarisch erstellt. Außer bei sehr kurzlebigen Systemen geht die Rechnung mit der Kosteneinsparung jedoch nicht auf (Bild 10.1).

Wird in einem Unternehmen, das seine Software-Entwicklung bisher nicht oder nur rudimentär dokumentiert hat, sorgfältiges Dokumentieren eingeführt, so ist folglich zu erwarten, dass die Produktivität in den frühen Phasen der Entwicklung absinkt und dafür in den späten Phasen sowie in der Nutzung steigt. Ferner darf eine Verbesserung der Produktqualität erwartet werden.

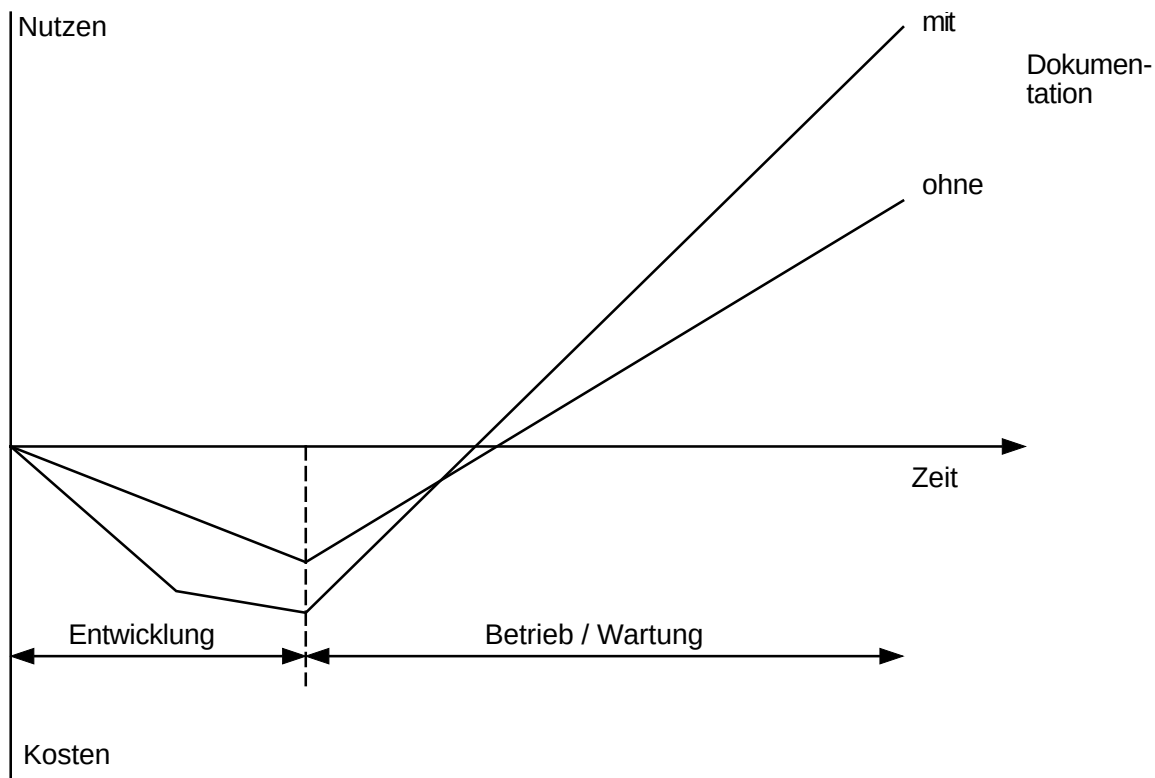


Bild 10.1. Wirtschaftlichkeit eines Software-Systems mit und ohne Dokumentation

Dokumentation ist nicht Selbstzweck. Es darf daher nur soviel wie unbedingt notwendig dokumentiert werden, dies aber sorgfältig und konsequent. Außerdem ist das notwendige Minimum schon recht viel. In der Regel wird nicht zuviel, sondern zuwenig dokumentiert.

10.2 Produktdokumente

Die Produktdokumentation muss folgende Aspekte eines Systems dokumentieren:

- die Anforderungen an das System
- das Konzept der Lösung
- die Einzelheiten der Lösung (Entwürfe und Realisierungen)
- die Montage der einzelnen Komponenten (Integration und Installation)

- die Planung von Tests und der Abnahme
- die Handhabung des Systems (Benutzerdokumentation).

Die *Anforderungsspezifikation* legt präzise, detailliert und (soweit wie möglich) nachprüfbar fest, was von dem zu entwickelnden System verlangt wird.

Das *Lösungskonzept* beschreibt die Architektur der Lösung. Dies umfasst insbesondere die Gliederung der Lösung in Komponenten (Teilsysteme, Aufteilung in Prozesse, Modularisierung), die Kommunikation zwischen diesen Komponenten, die Verteilung auf Software und Hardware sowie die Verteilung der Software auf die vorhandenen Ressourcen.

Entwürfe und *Programmcodes* beschreiben die Lösungsdetails (verwendete Algorithmen und Datenstrukturen). Entwürfe werden entweder separat vom Programmcodes dokumentiert oder sie sind in Form von ausführlichen Kommentaren im Programmcodes integriert.

Die *Testvorschriften* legen fest, welche Tests für die einzelnen Komponenten nach ihrer Fertigstellung durchzuführen sind und welche Tests nach welchem Integrationsschritt auszuführen sind.

Der in der *Abnahmevorschrift* beschriebene Abnahmetest bildet den formalen Abschluss einer Entwicklung. Der Auftraggeber überprüft mit diesem Test, ob das System die in der Anforderungsspezifikation gestellten Anforderungen erfüllt.

Der *Integrationsplan* beschreibt die Integration der einzeln fertiggestellten Komponenten zu einem in einer Testumgebung lauffähigen Gesamtsystem.

Die *Installationsanleitung* beschreibt, wie ein auf der Zielhardware lauffähiges System konfiguriert und auf der Zielhardware installiert wird.

Das *Benutzerhandbuch* enthält die Bedienungsanleitung für das System. Es beschreibt aus Benutzersicht, welche Funktionen das System bereitstellt und wie man es startet und bedient.

Zu einem eingebetteten System gibt es kein Benutzerhandbuch. Seine Benutzung wird im Rahmen der Bedienungsanleitung des übergeordneten Systems dokumentiert.

Ein *Glossar*, welches die verwendeten Begriffe und Abkürzungen erklärt, ist sowohl in der Entwicklung als auch nachher für die Benutzer sehr hilfreich. Ein solches Glossar sollte daher in jedem größeren Entwicklungsprojekt (projektbegleitend) erstellt werden.

10.3 Projektdokumente

Die wichtigen Projektdokumente sind der Projektplan, der Qualitätsplan und das Projektprotokoll.

Der *Projektplan* dokumentiert den geplanten Projektablauf, wobei den Sollvorgaben während der Projektabwicklung laufend die IST-Werte gegenübergestellt werden.

Der *Qualitätsplan* enthält die Vorgaben, die das Projekt betreffend Qualität zu beachten hat. Ist ein standardisierter Rahmen-Qualitätsplan vorhanden, sind projektspezifische Qualitätspläne häufig nicht erforderlich.

Das *Projektprotokoll* enthält alle Schriftstücke und Berichte (z.B. Änderungsanträge, Review- und Testprotokolle, Fortschrittsberichte, Sitzungsprotokolle), die im Verlauf der Projektabwicklung anfallen.

10.4 Dokumentverwaltung

Dokumente, die man nicht findet, wenn man sie braucht, oder solche, die nicht mehr aktuell sind, sind von zweifelhaftem Wert. Dokumente müssen daher der *Konfigurationsverwaltung* (vgl. Kapitel 11) unterworfen werden. Vor allem die folgenden drei Dinge sind wichtig.

Klassifikation: Leichtes Finden durch geordnetes Ablegen

- Jedes Dokument hat einen eindeutigen Namen.
- Alle Dokumente haben einheitlich gestaltete Deckblätter
- Jedes Blatt jedes Dokuments hat eine Kopfzeile mit folgenden Angaben: Projekt, Dokumentname (evtl. Kurzname), Versionsnummer / Änderungsindex, evtl. Datum, Kapitelnummer, Seitenzahl. Bei Dokumenten auf Datenträgern oder in Informationssystemen wird so verfahren, dass diese Informationen abrufbar sind, bzw. bei der Ausgabe auf Papier generiert werden.

Freigabewesen: Nur Freigegebenes gilt

Fertiggestellte Dokumente werden geprüft, gefundene Mängel werden behoben. Dann erst wird das Dokument freigegeben und verteilt. Von diesem Zeitpunkt an ist ein Dokument dem Zugriff seiner Ersteller entzogen. Änderungen sind nur noch über das Änderungswesen möglich.

Änderungswesen: Nur Aktuelles ist hilfreich

Das Änderungswesen hat zwei Ziele:

- Dokumente immer aktuell zu halten
- wildes, unkontrolliertes Ändern von Dokumenten zu verhindern.

Hierzu wird ein Prozess definiert, nach dem Änderungen ablaufen. In der Regel enthält er die Schritte: Antrag - Entscheid - Durchführung - Freigabe.

10.5 Dokumenterstellung

Dokumente entstehen *schritthaltend* mit der Entwicklung; Produktdokumente sind ein Bestandteil des zu entwickelnden Produkts. Ohne Dokumente sind weder eine vernünftige Projektführung noch eine geordnete Prüfung und Qualitätssicherung möglich. Das heute immer noch verbreitete „Hinterher“-Dokumentieren, welches dann oft dem Zeitmangel zum Opfer fällt, sollte endgültig der Vergangenheit angehören.

Hingegen kann es sinnvoll sein, zum Schluss einer Entwicklung Dokumente nochmals zu überarbeiten. Die Klarheit und die Konsistenz der Darstellung lässt sich im Nachhinein oft noch massiv verbessern. Der Aufwand für diese Verbesserungen lohnt sich, da ein guter Dokumentensatz die Pflege verbilligt.

Aufgaben

10.1 Nehmen Sie Stellung zu folgenden Aussagen:

- a) „Ich dokumentiere nicht, weil wir in unserem kleinen Projekt alles auch so im Griff haben.“
- b) „Ich dokumentiere keine Anforderungen, weil diese sich im Projektverlauf ja ohnehin ändern.“
- c) „Ich dokumentiere meinen Entwurf, sobald sich beim Testen zeigt, dass der Entwurf jetzt stabil ist.“

10.2 Warum müssen Dokumente der Konfigurationsverwaltung unterliegen?

11. Konfigurationsverwaltung

11.1 Grundlagen

„Ändern Sie noch eben schnell..." Die allzu einfache Möglichkeit, Software zu ändern, verursacht eine Menge von Problemen, zum Beispiel

- Codieren anhand der falschen Version des Entwurfs
- Paralleles, unkoordiniertes Ändern eines Moduls durch mehrere Personen
- Undokumentierte Schnellreparaturen an in Betrieb befindlicher Software.

Konfigurationsverwaltung ist das wichtigste Mittel zur Verhinderung bzw. zur Lösung solcher Probleme.

Definition 11.1. *Software-Konfigurationsverwaltung (Software-Konfigurationsmanagement, software configuration management).* Die Gesamtheit aller Verfahren zur eindeutigen Kennzeichnung der Konfiguration eines Software-Systems mit dem Zweck, den Aufbau und alle Änderungen dieser Konfiguration systematisch zu überwachen, die Konsistenz des Software-Systems sicherzustellen und die Möglichkeit der Rückverfolgung anzubieten.

Definition 11.2. *Software-Konfiguration.* Eine Menge zusammenpassender Software-Einheiten.

Definition 11.3. *Software-Einheit (software configuration item).* Der kleinste, im Rahmen der Konfigurationsverwaltung als atomar behandelte Baustein einer Konfiguration. Software-Einheiten werden nur als Ganzes registriert, freigegeben oder geändert. Software-Einheiten sind z.B. Programm-Module und Dokumente.

11.2 Kennzeichnung

Software-Einheiten haben eine eindeutige *Kennzeichnung*, bestehend aus einem Namen und einer Versionsnummer für das Änderungswesen (Bild 11.1). Die *Identität* einer Software-Einheit ist feststellbar, z.B. mit Prüfsummen.



Bild 11.1: Kennzeichnung von Software-Einheiten

11.3 Registrierung und Verwaltung

Die Software-Einheiten werden von einem *Software-Bibliothekar* registriert und verwaltet. Idealerweise erfolgt diese Verwaltung rechnergestützt mit Hilfe von Werkzeugen, welche Software-Einheiten, Konfigurationen und Releases in einer Datenbank ablegen und verwalten.

Nummer	Name	Typ	Ver	Prüfsumme	Status
...					
LOG 0021	Materialwesen	EntwDok	02	0873451-2	freigegeben
LOG 0027	Stückliste	Prog	03	0372538-1	freigegeben
LOG 0028	Verwendungsnachweis	Prog	02	0576927-6	in Prüfung
...					

Bild 11.2: Registrierung von Software-Einheiten

Von jeder Einheit können mehrere *Versionen* geführt werden. Im einfachsten Fall wird durch aufsteigende Versionsnummern deutlich gemacht, in welcher Reihenfolge die Versionen entstanden sind.

Im allgemeinen Fall wird zwischen *Revisionen* (entstehen durch Überarbeitung) und *Varianten* (haben gemeinsame Eigenschaften und in der Regel eine gemeinsame Vorgängerversion) unterschieden (Conradi und Westfechtel 1998).

11.4 Konfiguration und Release

Soll eine Konfiguration an Kunden ausgeliefert werden, so wird ein Release (vgl. Definition 3.10) gebildet (Bild 11.3).

Wesentliche Aufgaben in der Verwaltung von Releases sind die Buchführung darüber, welche Software-Einheiten dazugehören, wie diese voneinander abhängen und wie aus den Einheiten ein auslieferbares System generiert wird.

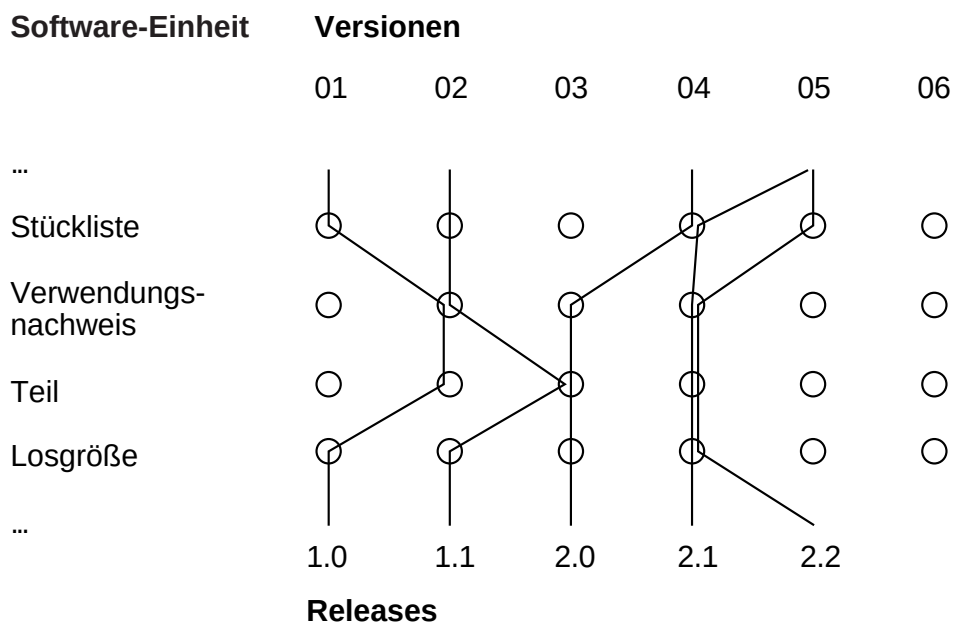


Bild 11.3: Eine Folge von Releases

11.5 Änderungswesen

Grundlage eines geordneten Änderungswesens sind getrennte Umgebungen für Entwicklung (Arbeitsumgebung) Verwaltung (Referenzumgebung) Test (Testumgebung) und operativen Einsatz (Produktionsumgebung) der Software-Einheiten (Bild 11.4).

Freie Änderungen sind nur in Arbeitsumgebungen gestattet. Für Software-Einheiten in der Referenzumgebung gilt ein strikt reglementiertes Änderungsprozedere. Änderungen in der Produktionsumgebung sind verboten.

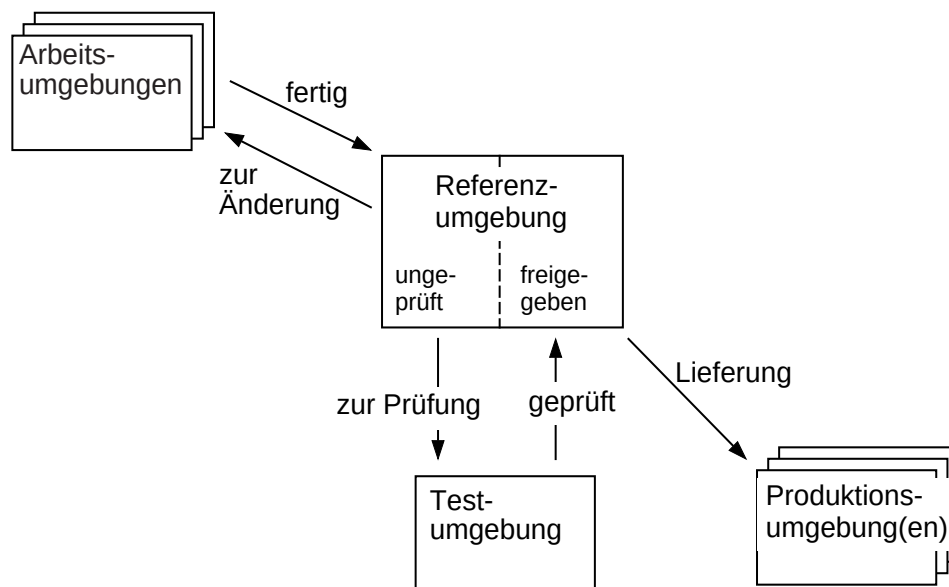


Bild 11.4: Umgebungen für Herstellung, Verwaltung und Nutzung von Software-Einheiten

Der *Änderungsstand* einer Software-Einheit muss jederzeit nachweisbar sein, beispielsweise durch Führung von Formularen, auf denen Änderungen beantragt, bewilligt und ihre Ausführung nachgewiesen werden. Der Ablauf einer Änderung folgt einem festgelegten Prozess (Bild 11.5)

11.6 Behandlung von Problemmeldungen

Grundlage für die Behandlung von Problemen, die beim Verwender eines Software-Systems auftreten, ist ein organisiertes Problemmeldungswesen. Da die Verwender nicht wissen können, ob ihr Problem durch einen Software-Fehler oder beispielsweise durch Unkenntnis oder eine Fehlbedienung verursacht ist, wird bewusst nicht von Fehlermeldungen, sondern von Problemmeldungen gesprochen. Benötigt werden unter anderem ein Problemmeldungsformular (siehe Anhang) und ein geordneter Bearbeitungsablauf.

Möglicher Bearbeitungsablauf:

- Registrierung eingegangener Problemmeldungen
- Analyse der Meldung
- Vorläufige Antwort
- Fehlerbehebung
- Abschließende Antwort

- Abschluss und Ablage der Problemmeldung
- Auslieferung von neuem Release

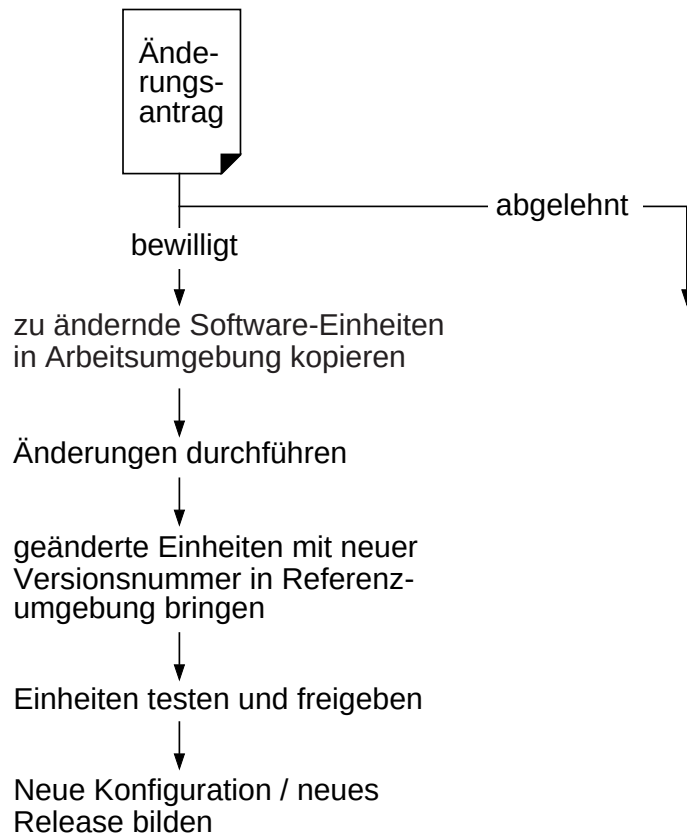


Bild 11.5: Ablauf einer Änderung

Zitierte und ergänzende Literatur

- Babich, W. A. (1986). *Software Configuration Management*. Reading, Mass.: Addison-Wesley.
- Bersoff, E.K., V.P. Henderson, S.G. Siegel (1980). *Software Configuration Management*. Englewood Cliffs, N.Y.: Prentice-Hall.
- Conradi, R., B. Westfechtel (1998). Version Models for Software Configuration Management. *ACM Computing Surveys* **30**, 2 (June 1998). 232-282.
- Frühauf, K., J. Ludewig, H. Sandmayr (1991). *Software-Projektmanagement und -Qualitätssicherung*. Zürich: Verlag der Fachvereine und Stuttgart: Teubner.
- IEEE (1983). *IEEE Standard for Software Configuration Management Plans*. ANSI/IEEE Std 828-1983. IEEE Computer Society Press.
- IEEE (1987). *IEEE Guide to Software Configuration Management*. ANSI/IEEE Std 1042-1987. IEEE Computer Society Press.

Anhang: Problemmeldungsformular

Problemmeldung		Nr.										
Verfasser												
Name _____		Datum _____										
Firma _____		Telefon / Fax / E-mail _____										
Adresse _____												
Betrifft		Problem ist										
☐ Produkt _____ Release/Version _____ ☐ Leistung _____ ☐ anderes _____ Verwendete Hardware _____ Betriebssystem _____		<table style="width: 100%; border: none;"> <tr> <td></td> <td style="text-align: center;">ja</td> <td style="text-align: center;">nein</td> </tr> <tr> <td>reproduzierbar</td> <td style="text-align: center;">☐</td> <td style="text-align: center;">☐</td> </tr> <tr> <td>umgehbar</td> <td style="text-align: center;">☐</td> <td style="text-align: center;">☐</td> </tr> </table> Problem betrifft ☐ Programme ☐ Unterlagen ☐ Leistungen Antwort erwartet bis _____			ja	nein	reproduzierbar	☐	☐	umgehbar	☐	☐
	ja	nein										
reproduzierbar	☐	☐										
umgehbar	☐	☐										
Problembeschreibung ☐ Problembeschreibung in Beilage 												
zu treffende Maßnahmen		Klassifizierung der Maßnahmen										
		Fehlerbehebung ☐ Anpassung ☐ Erweiterung ☐ Beratung/Info ☐ Schulung ☐										
Verantwortlicher Sachbearbeiter												
Name _____		Datum _____										
		Visum _____										
Zwischenbescheid an Kunde												
(erforderlich, wenn Meldung nicht bis zum vom Kunden erwarteten Termin erledigt werden kann)		Datum _____										
		Visum _____										
Problem erledigt und Kunde informiert												
Name _____		Datum _____										
		Visum _____										

12. Produktivitätsfaktoren

12.1 Werkzeuge

12.1.1 Allgemeines

Im Informatik-Sprachgebrauch sind *Werkzeuge* rechnergestützte Hilfsmittel für die Entwicklung von Software. Mit dem Einsatz von Werkzeugen werden primär folgende Ziele verfolgt:

- Entlastung der Entwickler von Routinearbeiten
- Bearbeitung der Sprachen (graphisch oder textuell), mittels derer die Entwickler ihre Probleme und Lösungen formulieren
- Unterstützung des Einsatzes von Entwicklungsmethoden
- Vereinfachung von Änderungen.

Werkzeuge sind jedoch – entgegen einem weit verbreiteten Glauben – keine Wunderwaffen:

- sie steigern die Produktivität nicht um (dezimale) Größenordnungen (es sei denn, man hätte vorher die Bits noch einzeln ins Silizium gemeißelt)
- sie ersetzen das eigene Denken und sorgfältige Arbeiten nicht (d.h. sie sind *nicht* intelligent, weder künstlich noch natürlich)
- sie führen zwar in den meisten Fällen zu verbesserter Qualität der Produkte, machen aber das Qualitätsmanagement (Reviews, Tests, etc.) nicht überflüssig.

Ein isolierter Einsatz von Werkzeugen bringt in der Regel wenig oder kann sogar kontraproduktiv sein. Werkzeuge müssen als eine Komponente in einer Gesamtstrategie für rechnergestütztes Software Engineering aufgefasst werden (Bild 12.1; Glinz 1990). Die Sache (Software Engineering) und das Mittel (computergestützt) dürfen in ihrer Wichtigkeit nicht miteinander verwechselt werden.

Seit etwa 1986 werden Werkzeuge (vor allem solche für Spezifikation und Entwurf) von Software häufig mit dem Schlagwort *CASE* (computer aided software engineering) bezeichnet.

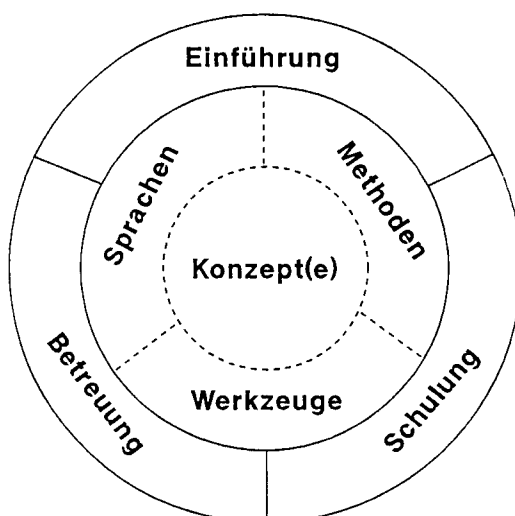


Bild 12.1: Elemente einer Strategie für computergestütztes Software Engineering

Produktivitätssteigerungen von 20-50% sind durch geeigneten Werkzeugeinsatz durchaus erreichbar. Dieser Produktivitätsgewinn stellt sich jedoch nicht sofort ein. Wird ein Werkzeug neu eingeführt, so müssen die Mitarbeiter intensiv geschult werden und müssen sich an den Umgang mit dem Werkzeug gewöhnen. Als Konsequenz sinkt die Produktivität erst einmal, anstatt zu steigen. Noch schlimmer ist die Situation bei Werkzeugen für Spezifikation und Entwurf von Software. Da beide Bereiche bei manueller Arbeit traditionell vernachlässigt werden, hat der Werkzeugeinsatz zur Folge, dass die Spezifikation und Konzipierung deutlich länger dauern als vorher. Der Gewinn macht sich erst in einer verkürzten Realisierung und noch später in niedrigeren Pflegekosten bemerkbar.

Werkzeugeinsatz ist daher keine Maßnahme, die sofort wirksam wird, sondern eine *Investition*, die sich erst mittelfristig (dann aber umso deutlicher) auszahlt.

Nicht zu vernachlässigen ist neben dem Produktivitätsgewinn ein u.U. massiver Qualitätsgewinn bei der mit Werkzeughilfe entwickelten Software.

12.1.2 Software-Produktionsumgebungen

Die vollständige Software Engineering-Umgebung, welche alle Aktivitäten von der Formulierung der ersten Anforderungen bis zur Systemabnahme durchgehend und konsistent unterstützt, ist zwar schon vielfach postuliert und skizziert worden, aber es gibt sie immer noch nicht zu kaufen.

Die heutige Situation ist geprägt von einer Vielfalt von Systemen mit unterschiedlichen Schwerpunkten, welche jeweils einen Teil der Software Engineering-Aktivitäten gut, einen Teil halbwegs und einen Teil überhaupt nicht unterstützen.

12.1.3 Klassifizierung von Werkzeugen

Die heute verfügbaren Werkzeuge lassen sich grob in folgende Gruppen klassifizieren:

Editoren

Editoren rationalisieren die Schreib- und Zeichenarbeiten. Struktureditoren und syntaxgesteuerte Editoren beschleunigen die Dokumentationserstellung und die Programmierung.

Spezifikations- und Entwurfssysteme

Diese Systeme unterstützen das Erstellen von Anforderungen und von Konzepten (Architekturentwürfen) für Systeme vorzugsweise durch graphische Darstellungen und durch geeignete Speicherung der Information in einer Datenbank.

Programmentwurfssysteme

In diese Kategorie fallen Werkzeuge zur Bearbeitung von Struktogrammen, Pseudocode, Jackson-Diagrammen, Programmablaufplänen, etc.

Compiler, Browser und Programmierumgebungen

Auch die gewöhnlichen Compiler für Programmiersprachen sind Werkzeuge. Programmierumgebungen vereinigen Editoren, Compiler, Browser (Software zum Durchstöbern der Arbeitsumgebung und von Software-Bibliotheken), evtl. Interpreter, Laufzeitumgebung, Debugger und evtl. ein Konfigurationsverwaltungssystem zu einem Satz aufeinander abgestimmter Werkzeuge für die Programmierung.

Programmgeneratoren

Programmgeneratoren sind Übersetzer, welche aus einer Hochsprache (v.a. Datenbanksprachen, sogen. Sprachen der 4. Generation) Programme in einer niedrigeren Sprache (v.a. COBOL und Job-Steuersprachen) erzeugen.

Mess- und Testwerkzeuge

Werkzeuge dieser Kategorie dienen dazu, interessierende Merkmale eines Programms (zum Beispiel Größe oder Komplexitätsattribute) zu messen und zu analysieren, Programme für die Ausführung und Auswertung von Tests zu instrumentieren, Testfälle, Testdaten und Testumgebungen zu generieren sowie Testergebnisse auszuwerten.

Konfigurationsverwaltungssysteme

Konfigurationsverwaltungssysteme vereinfachen das Verwalten verschiedener Versionen von Programmen und Dokumenten und ermöglichen das Generieren von Konfigurationen, d.h. das automatische Zusammensetzen eines Systems aus einer Reihe von Programmen und Dokumenten, unter Umständen mit entsprechender Vorverarbeitung wie Compilieren, Binden, Sortieren, etc.

12.1.4 Planung des Werkzeugeinsatzes

Wenn man Werkzeuge für die Software-Entwicklung erfolgreich einführen will, muss man sich zunächst über folgende Fragen im Klaren sein:

- Welche Aktivitäten sollen die Werkzeuge unterstützen bzw. automatisieren?
- Wie wirtschaftlich ist der Werkzeugeinsatz? (Ein syntaxgesteuerter Editor beispielsweise, welcher den Aufwand für die Codierung um 25% vermindert, bringt nur 5% Verminderung des Gesamt-Projektaufwands, wenn man 20% Aufwandsanteil für die Codierung veranschlagt)
- Welche Entwicklungskonzepte, und damit verbunden, welche Methoden und Sprachen sollen eingesetzt werden?
- Ist die Schulung geregelt (Freistellung der Mitarbeiter, Finanzierung, Verfügbarkeit von Kursen)?
- Wie sieht die Einführungsstrategie aus?
- Ist die Betreuung der Werkzeugverwender sichergestellt?

12.2 Mehrfachverwendung von Software

Die Bedeutung von Mehrfachverwendung für die Produktivität wurde in Kapitel 3.4 diskutiert.

12.3 Die Rolle der Menschen

Software wird von Menschen gemacht. Die Software-Leute sind daher neben der Software-Mehrfachverwendung der entscheidende Produktivitätsfaktor in der Software-Entwicklung.

Aspekte der Psychologie und der Menschenführung spielen somit eine wichtige Rolle in der Software-Entwicklung. Wesentliche Gedanken dazu enthalten die Bücher von Brooks (1995), DeMarco und Lister (1991) und Weinberg (1971). In diesem Kapitel werden nur wenige Grundsätze und Regeln beschrieben.

12.3.1 Regeln über Menschen in der Software-Entwicklung

Produktivität

Die Schwankungsbreite der individuellen Produktivität von Software-Entwicklern ist sehr groß. Ein von Sackman (1968) durchgeführtes Experiment ergab Schwankungsbreiten in der Produktivität von ca. 20:1. Diese Daten sind zwar schon recht alt, aber es gibt keinen Grund zur Annahme, dass sich seither etwas Grundlegendes verändert hat. Boehm (1987) gibt eine Schwankungsbreite von rund 4 an, wobei er allerdings nicht die besten und die schlechtesten Entwickler vergleicht, sondern Gruppen, die einerseits unter 15% und andererseits über 90% auf einer relativen Fähigkeitsskala liegen.

Disponibilität

Personal ist kein beliebig disponierbarer Produktionsfaktor. Drei Regeln sind zu beachten:

- Personalbestände in einem Projekt können nur langsam auf- oder abgebaut werden. Es ist nicht ohne Schaden möglich, kurzfristig massive Veränderungen im Personalbestand eines Projekts vorzunehmen.
- Zu einem gegebenen Entwicklungsaufwand gibt es eine optimale Gruppengröße für die Bearbeitung und damit eine optimale Durchlaufzeit. Die Rechnung, die Durchlaufzeit durch den Einsatz von mehr Entwicklern zu verkürzen, geht nur in sehr engen Grenzen auf.
- Das Aufstocken des Personalbestands in einem verspäteten Projekt führt zu noch mehr Verspätung (Gesetz von Brooks; Brooks 1995).

Arbeitsverteilung

Software-Entwickler schreiben nicht nur Programme. Eine Studie der Bell Labs ergab, dass der im Hinblick auf Codeerzeugung produktive Anteil an der Arbeit eines Programmierers nur 13% beträgt (Bild 10.1).

Programme schreiben	13%	
Programme und Handbücher lesen	16%	
Arbeitsbezogene Kommunikation	32%	(!)
Persönliches	13%	
Ausbildung	6%	
Post	5%	
Diverses	15%	Studie der Bell Labs, 1964, zitiert aus Fairley (1985)

Bild 12.2: Verteilung der Arbeitszeit eines Programmierers

Gruppengröße

Jede Entwicklergruppe benötigt einen Teil ihrer Zeit für Organisation und Kommunikation innerhalb der Gruppe. Mit zunehmender Personenzahl nimmt dieser nicht produktive Aufwand überproportional zu (vgl. dazu Bild 1.6).

12.3.2 Emotionales und Rationales in der Software-Entwicklung

Emotionale Einstellungen der an Software-Entwicklung Beteiligten haben erhebliche Einflüsse auf das, was in einer Entwicklungsabteilung tatsächlich geschieht. Der Einfluss des Emotionalen ist umso stärker, je weniger er den Beteiligten bewusst ist. (Glinz 1988).

12.3.3 Arbeitsumgebung

Da Software von Menschen gemacht wird, haben Ausbildung und Arbeitsumgebung dieser Menschen einen erheblichen Einfluss auf den Produktionsprozess. Im Buch von DeMarco und Lister (1991) werden verschiedene Aspekte dieses Problems betrachtet.

Aufgaben

- 12.1** Eine Software-Entwicklungsabteilung schreibt rote Zahlen. Anstatt ein besonders sparsames Budget vorzulegen, beantragt die Abteilungsleiterin zum Erstaunen der Geschäftsleitung eine Investition von 100.000 Franken zur Modernisierung der Arbeitsplätze der Entwicklerinnen und Entwickler sowie für Ummöblierungen und Gardinen(!). Was könnte sich die Frau dabei gedacht haben?
- 12.2** Ein Unternehmen will die Produktivität seiner Software-Entwicklungsabteilung verbessern. Beschreiben Sie, welche Rolle Werkzeuge dabei spielen können.

Zitierte und weiterführende Literatur

- Boehm, B. (1987). Improving Software Productivity. *IEEE Computer* **20**, 9 (Sept. 1987). 43-57.
- Brooks, F.P. (1995). *The Mythical Man Month. Essays on Software Engineering*. Anniversary Edition Reading, Mass., etc.: Addison-Wesley. (Neuausgabe des Originals von 1975)
- DeMarco, T., T. Lister (1991). *Wien wartet auf Dich! Der Faktor Mensch im DV-Management*. München-Wien: Hanser.
- Fairley, R. (1985). *Software Engineering Concepts*. New York, etc.: McGrawHill.
- Glinz, M. (1988). Emotionales und Rationales im industriellen Software Engineering. *Technische Rundschau* 20/88, 78-81.
- Glinz, M. (1990) Warte nicht auf bessere Zeiten - Methoden und Werkzeuge in der Software-Entwicklung. *Technische Rundschau* 35/90, 70-75
- IEEE (1992). Themenheft Tools Assessment. *IEEE Software* **9**, 3 (May 1992).
- Ludewig, J. (1991). Software Engineering und CASE - Begriffsklärung und Standortbestimmung. *it (Informationstechnik)* **33**, 3. 112-120.
- Sackman, H. et al. (1968). Exploratory Experimental Studies Comparing Online and Offline Programming Performance. *Communications of the ACM* **11**, 1 (Jan. 1968).
- Weinberg, G.M. (1971). *The Psychology of Computer Programming*. New York: Van Nostrand Reinhold.

Kommentiertes Literaturverzeichnis von Lehrbüchern und wichtigen Zeitschriften-Artikeln über Software Engineering

Balzert, H. (1996). *Lehrbuch der Software-Technik: Software-Entwicklung*. Heidelberg: Spektrum Akademischer Verlag. 1009 p. + CD-ROM.

Balzert H. (1998). *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. 769 p. + CD-ROM.

Enzyklopädische Fülle von Material. Präsentation überladen mit Details. Konzepte, Zusammenhänge und die Bedeutung der einzelnen Themen fehlen oder gehen unter. Behandelt auch viele nicht zum Kern des Software Engineerings gehörende Themen. Als Lehrbuch weniger geeignet.

Boehm, B. (1981). *Software Engineering Economics*. Englewood Cliffs, N.J.: Prentice Hall. 767 p.

Ein Standardwerk. Fundgrube für empirische und quantitative Aussagen. Als Lehrbuch weniger geeignet.

Boehm, B. (1983). Seven Basic Principles of Software Engineering. *Journal of Systems and Software* **3**, 3-24.

Ein klassischer Zeitschriften-Artikel, welcher v.a. den Management-Aspekt des Software Engineerings in sieben Prinzipien zusammengefasst darstellt.

Brooks, F.P. (1995). *The Mythical Man Month. Essays on Software Engineering*. Anniversary Edition Reading, Mass., etc.: Addison-Wesley.

Unterhaltsame Beschreibung von Fehlern im Software Engineering und ihren Ursachen. Neuausgabe des Originals von 1975 mit zwei neuen Kapiteln. Eine Pflichtlektüre für alle, die sich für Software Engineering interessieren.

Brooks, F.P. (1987). No Silver Bullet. Essence and Accidents of Software Engineering. *IEEE Computer* **20**, 4. 10-19.

Klassischer Zeitschriftenartikel über Einfaches und Schwieriges (und die Nichtexistenz von Patentlösungen) im Software Engineering.

Denert, E. (1991). *Software Engineering*. Berlin, etc.:Springer. 452 p.

Kein Buch über Software Engineering allgemein, sondern ein Lehrbuch über Entwicklung von Informationssystemen nach einer vom Verfasser geschaffenen Methode. Für dieses Teilgebiet ein sehr gutes, praxisnahes Buch; als allgemeine Einführung in Software Engineering nicht geeignet.

Dumke, R. (1993). *Modernes Software Engineering: Eine Einführung*. Braunschweig/ Wiesbaden: Vieweg. 305 p.

Sammelsurium verschiedener Themen aus dem Software Engineering. Nicht empfehlenswert.

Fairley, R. (1985). *Software Engineering Concepts*. New York: McGrawHill. 364 p.

Kompakte, sehr gute Einführung in die Thematik. Leider seit Erscheinen nicht mehr revidiert, daher nicht mehr auf dem neuesten Stand, z.B. keine Aussagen über Objektorientierung. Trotzdem immer noch lesenswert.

Frühauf, K., Ludewig, J., Sandmayr, H. (1999). *Software-Projektmanagement und -Qualitätssicherung*. Dritte, überarbeitete Auflage. Zürich: vdf.

Einführung in zwei zentrale Teilgebiete des Software Engineerings in der Art einer Fibel. Empfehlenswert.

Ghezzi C., M. Jazayeri, D. Mandrioli (1991). *Fundamentals of Software Engineering*. Englewood Cliffs, N.J.: Prentice Hall. 573 p.

Ansprechende Darstellung. Gewicht mehr auf technischer Seite des SE, Beschreibung nicht nur von Ausprägungen, sondern auch der zugrundeliegenden Konzepte. Empfehlenswert.

Gilb, T. (1988). *Principles of Software Engineering Management*. Reading, Mass.: Addison Wesley. 442 p.

Baut auf drei Ideen auf: Messbare Anforderungen, evolutionäre Entwicklung, Prüfung mit Inspektionen. Keine vollständige Einführung ins Thema Software Engineering. Populistischer Stil in der Darstellung.

Jalote, P. (1991). *An Integrated Approach to Software Engineering*. Second edition. Berlin, etc.: Springer. 497 p.

Konventionelle Einführung anhand eines Durchgangs durch den Software Lebenslauf. In der zweiten Auflage sind zu den strukturierten Methoden die objektorientierten hinzugekommen. Durchgehende Fallstudie.

Johnson, J.R. (1991). *The Software Factory: Managing Software Development and Maintenance*. 2nd edition. Wellesley, Mass.: QED Information Sciences. 277 p.

Etwas oberflächlich/populärwissenschaftlich; Themenauswahl stark auf Entwicklung betrieblicher Informationssysteme bezogen. Stärke: Enthält in den Anhängen Zahlen zu Messungen im Software-Entwicklungsprozess.

Jones, G.W. (1990). *Software Engineering*. New York: John Wiley & Sons. 480 p.

Vom Inhalt her an sich kein schlechtes Buch. Insgesamt aber ist das Material zu breit und zu oberflächlich dargestellt. Schlechtes, das Lesen behinderndes Layout.

Kahlbrandt, B. (1998). *Software-Engineering*. Springer, Berlin, etc.

Etwas unsystematische Mischung aus Lehrbuch über Software Engineering und Einführung in die UML (Unified Modeling Language).

Kimm, R. et al. (1979). *Einführung in Software Engineering*. Berlin: Walter de Gruyter. 306 p.

Zum Zeitpunkt seines Erscheinens eines der besten Lehrbücher zum Thema. Aus heutiger Sicht teilweise hoffnungslos veraltet, teilweise mit erheblichen Lücken. Heute als Lehrbuch leider nicht mehr verwendbar.

Lamb, D.A. (1988). *Software Engineering: Planning for Change*. Englewood Cliffs, N.J.: Prentice Hall. 298 p.

Knappe, teilweise oberflächliche Behandlung einer Sammlung von SE-Themen. Schwerpunkt auf Spezifikation und Entwurf. Grundprobleme und Zusammenhänge werden nicht verdeutlicht.

Ludewig, J. (Hrsg.) (1991). *Software- und Automatisierungsprojekte - Beispiele aus der Praxis*. Stuttgart: Teubner. 227 p.

Kein Lehrbuch, sondern eine Zusammenstellung von Erfahrungsberichten über abgewinkelte Software-Projekte aus der Praxis. Besonders für Studierende ein guter Einblick in Anwendung bzw. Nicht-Anwendung von Software Engineering-Prinzipien in der Praxis.

Marco, A. (1990). *Software Engineering*. Englewood Cliffs, N.J.: Prentice Hall. 544 p.

Darstellung etwas breit, Themenauswahl im Detail z.T. etwas eigenwillig, keine klare und systematische Darstellung der Grundprinzipien.

Mayrhauser, A. von (1990). *Software Engineering - Methods and Management*. San Diego: Academic Press. 864 p.

In Inhalt und Aufbau von mäßiger Qualität, teilweise veraltet, viel zu umfangreich: nicht empfehlenswert.

McDermid, J. (1991). *Software Engineer's Reference Book*. Oxford: Butterworth-Heinemann. ca. 1200 p.

Behandelt alle Gebiete der Informatik mit Bezug zum Software Engineering einschließlich der Grundlagen- und Randgebiete in knappen Übersichtsartikeln in der Art einer Enzyklopädie, aber nach Sachgebieten, nicht nach Stichworten geordnet. Nachschlagewerk, kein Lehrbuch.

Nagl, M. (1990). *Softwaretechnik: Methodisches Programmieren im Großen*. Berlin, etc.: Springer. 387 p.

Beschreibt im Wesentlichen nur das Problem des Software-Entwurfs; dieses aber sorgfältig und gründlich. Keine Einführung in Software Engineering.

Pagel, B.U., H.W. Six (1994). *Software Engineering - Band 1: Die Phasen der Software-Entwicklung*. Bonn, etc.: Addison-Wesley. 893 p.

Beschreibt in diesem ersten Band ausschließlich die Entwicklungstätigkeiten. Diese sind recht breit dargestellt.

Pfleeger, S.L. (1991). *Software Engineering: The Production of Quality Software*. 2nd edition. New York: Macmillan Publishing. 517 p.

Leitfadenartig aufgebaut, Auswahl und Darstellung vor allem im Detail oberflächlich, unsystematisch und teilweise veraltet.

Pomberger, G., G. Blaschek (1993). *Grundlagen des Software Engineering: Prototyping und objektorientierte Software-Entwicklung*. München: Hanser. 337 p.

Beschreibt die Arbeit in den einzelnen Entwicklungsphasen, ca. ein Drittel des Buchs behandelt Objektorientierung, wenig über den Management-Aspekt des Software Engineerings.

Pressman, R.S. (1992). *Software Engineering - A Practitioner's Approach*. Third Edition. New York, etc.: McGrawHill. 793 p.

Ansprechende und praxisorientierte, aber sehr umfangreiche Darstellung des Themas. Vor allem die technischen Aspekte sind sehr breit dargestellt.

Schach, S.R. (1996). *Classical and Object-Oriented Software Engineering*. Third edition. Chicago, etc: IRWIN. 603 p.

Gute Darstellung der Entwicklungsphasen. Objektorientierung wurde in der dritten Auflage hinzugefügt (die zweite Auflage hieß noch schlicht "Software Engineering"), was der Darstellung teilweise noch anzumerken ist. Management-Aspekte des Software Engineerings werden eher oberflächlich behandelt. Sehr gute Verweise auf weiterführende Literatur.

Schach, S.R. (1999). *Classical and Object-Oriented Software Engineering with UML and Java*. Fourth edition. Boston: WCB/McGraw-Hill. 616 p.

Teilweise reorganisierte, leicht erweiterte und aktualisierte Fassung von Schach (1996). Beispiele neu mit UML und Java. *Keine* Einführung in UML bzw. Java.

Sommerville, I. (1995). *Software Engineering*, Fifth Edition. Reading, Mass.: Addison Wesley. 742 p. (deutsche Übersetzung der zweiten Auflage ebenfalls bei Addison Wesley, 1987).

Umfassende Darstellung des Themas. In der Fülle ist jedoch das Wesentliche nicht mehr gut zu erkennen. Darstellung trotz ihres Umfangs zum Teil oberflächlich. In vielen Bibliotheken ist auch die vierte Auflage von 1992 noch greifbar. Deutsche Ausgabe in manchen Punkten nicht mehr aktuell, Management-Aspekt zuwenig gewichtet.

Suhr, R., R. Suhr (1993). *Software Engineering: Technik und Methodik*. München, Wien: Oldenbourg. 415 p.

Darstellung verschiedener, schwergewichtig technischer Aspekte des Software Engineerings. Ansprechend im Aufbau, leider in der Ausführung vieler Themen unbefriedigend.

van Vliet, H. (1993). *Software Engineering: Principles and Practice*. Chichester: John Wiley & Sons. 558 p.

Übersicht über alle Themengebiete des Software Engineerings, vielfach jedoch sehr oberflächlich. Die dargestellten Techniken sind teilweise veraltet, neuere fehlen.

Zehnder, C.A. (1991). *Informatik-Projektentwicklung*. 2. Auflage. Zürich: Verlag der Fachvereine und Stuttgart: Teubner. 309 p.

Konzentriert sich auf den Aspekt der Software-Entwicklung nach einem bestimmten Entwicklungsmodell. Beschreibt vor allem, was zu tun ist; wenig Angaben zur Methodik.